

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re *Inter Partes* Review of:

Inventor: Soltis

U.S. Patent No. 6,871,264

Issued: Mar. 22, 2005

For: System and Method for
Dynamic Processor Core and
Cache Partitioning on Large-
Scale Multithreaded,
Multiprocessor Integrated
Circuits

IPR2025-00418

**DECLARATION OF DR. BRUCE JACOB
IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW
OF U.S. PATENT NO. 6,871,264**

TABLE OF CONTENTS

I. Overview	5
A. Introduction	5
B. Qualifications	5
II. Legal Considerations.....	10
A. Level of Skill in the Art	10
B. Claim Construction	10
C. Identifications of How the Claims Are Unpatentable	11
IV. Technical Background	12
A. Multicore Processors	12
B. Processor Caches and Cache Hierarchies	17
1. General Cache Organization and Operation	18
2. Multilevel Caches	20
V. Overview of U.S. Patent 6,871,264	37
A. Patent Disclosure	37
B. The Claims of the '264 Patent	44
VI. Overview of the Prior Art	46
A. United States Patent 6,725,336 by Rajasekhar Cherabuddi ("Cherabuddi")	46
B. United States Patent 6,604,174 to Alvar Dean, et al. ("Dean")	54
C. United States Patent 6,381,682 to Karen Noel, et al. ("Noel")	60
VII. Analysis of the Claims of U.S. Patent 6,871,264	64
A. Claims 1–3, 5, and 8–10 Are Rendered Obvious by Cherabuddi	64

1.	<i>Claim 1</i>	64
2.	<i>Claim 2</i>	86
3.	<i>Claim 3</i>	90
4.	<i>Claim 5</i>	95
5.	<i>Claim 8</i>	129
6.	<i>Claim 9</i>	137
7.	<i>Claim 10</i>	142
B.	The Combination of Cherabuddi and Dean.....	146
C.	Claims 4–5 and 8–10 Are Rendered Obvious by the Combination of Cherabuddi and Dean	161
1.	<i>Claim 4</i>	161
2.	<i>Claim 5</i>	170
3.	<i>Claim 8</i>	184
4.	<i>Claim 9</i>	188
5.	<i>Claim 10</i>	191
D.	The Combination of Cherabuddi and Noel.....	196
E.	Claim 11 Is Rendered Obvious by the Combination of Cherabuddi and Noel.....	213
1.	<i>Claim 11</i>	213
	References	221

EXHIBIT LIST

No.	Description
1001	U.S. Patent No. 6,871,264
1002	U.S. Patent No. 6,725,336 ("Cherabuddi")
1003	U.S. Patent No. 6,604,174 ("Dean")
1004	U.S. Patent No. 6,381,682 ("Noel")
1005	Declaration of Dr. Bruce Jacob
1006	Curriculum Vitae of Dr. Bruce Jacob
1007	File History of U.S. Patent No. 6,871,264
1008	Interim Procedure for Discretionary Denials in AIA Post-Grant Proceedings with Parallel District Court Litigation by Director Katherine K. Vidal
1009	First Amended Docket Control Order entered in Valtrus Innovations Ltd. v. SAP Am. Inc., No. 2:24-cv-21, ECF 65 (E.D. Tex. Jan. 15, 2024)
1010	SAP America, Inc. and SAP SE's Motion to Dismiss for Lack of Standing Pursuant to Rule 12(b)(1) and Rule 12(b)(6)
1011	SAP America, Inc. and SAP SE's Motion to Reconsider Partial Denial of SAP's Motion to Dismiss and Grant of Valtrus's Motion for Leave to Amend Complaint (ECF 83)
1012	Valtrus and KPI's Complaint for Patent Infringement and Jury Demand, filed in Valtrus Innovations Ltd. v. SAP Am, Inc., No. 2:24-cv-533, ECF 1 (E.D. Tex. July 12, 2024)
1013	Valtrus and KPI's Statement Regarding Valtrus II (Case No. 2:24-CV-533-JRG), filed in Valtrus Innovations Ltd. v. SAP Am, Inc., No. 2:24-cv-21, ECF 85 (E.D. Tex. Nov. 15, 2024)
1014	John L. Hennessy and David A. Patterson, <i>COMPUTER ARCHITECTURE: A QUANTITATIVE APPROACH</i> (5th ed. 2012)
1015	Bruce Jacob, Spencer W. Ng, and David T. Wang, <i>MEMORY SYSTEMS: CACHE, DRAM, DISK</i> (2008)
1016	Daniel J. Sorin, Mark D. Hill, David A. Wood, <i>A PRIMER ON MEMORY CONSISTENCY AND CACHE COHERENCY</i> (2011)
1017	Kunle Olukotun, et al., <i>The Case for a Single-Chip Multiprocessor</i> , STANFORD UNIVERSITY: COMPUTER SYSTEMS LABORATORY (1996)
1018	Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun, <i>A Single-Chip Multiprocessor</i> , STANFORD UNIVERSITY (Sept. 1997)
1019	Alan Jay Smith, <i>Cache Memories</i> , 14 <i>COMPUTING SURVEYS</i> 3 (Sept. 1982)

I. OVERVIEW

A. Introduction

(1) I have been retained by Petitioner SAP to provide my opinions concerning the validity of claims in U.S. Patent No. 6,871,264 (“the ’264 patent”). I am being compensated for my time in connection to this matter. My compensation is not in any way contingent upon the opinions that I express or the outcome of this proceeding or any related litigation. I have no financial or personal interest in the outcome of this proceeding or any related litigation.

B. Qualifications

(2) In 1988, I earned the *Ars Baccalaureate* degree in mathematics from Harvard University in Cambridge, Massachusetts. In 1995 and 1997, respectively, I earned M.S. and Ph.D. degrees, both in Computer Science & Engineering, from the University of Michigan in Ann Arbor, Michigan. In 2020, I was named a Fellow of the IEEE (Institute of Electrical and Electronics Engineers) “for contributions to computer memory design and analysis.”

(3) From 1997 to 2003, I was an Assistant Professor in the Department of Electrical & Computer Engineering at the University of Maryland at College Park, Maryland. I was promoted to Associate Professor and was granted tenure in 2003.

In 2007, I was promoted to Director of Computer Engineering, a position I held until my sabbatical beginning Fall 2010. In the Spring of 2011, I was promoted to full Professor. In 2022, I joined the United States Naval Academy in Annapolis Maryland, where I am a Professor of Cyber Science and study the security of memories and computer hardware systems.

(4) For the past twenty-five years I have designed high-performance semiconductor memory systems and computer-systems architectures for both industry and government. For example, I helped Micron design their Hybrid Memory Cube architecture, redesigned Cray's memory controller for their Black Widow memory system, helped Northrop Grumman design a memory-system interconnect for their experimental ultra-low-power datacenter, designed a high-performance memory system for the 1024-core Teraflux chip funded by the European Commission, and have collaborated with the Departments of Energy and Defense on the design of their next-generation supercomputers. My group also fabricated semiconductor memory chips and a CPU chip as part of an early DOD research program on die stacking using through-silicon vias.

(5) I received the National Science Foundation CAREER Award in 1999 for my work in semiconductor memory devices and memory systems. At Maryland, I was decorated seven times as a Clark School "Rainmaker" (renamed "Research

Leader” in 2006) for my research program, being one of the top 100 researchers and fundraisers in the University.

(6) I have published two books on computer memory systems: Memory Systems: Cache, DRAM, Disk (Morgan Kaufmann) 2007; and The Memory System (You Can’t Avoid It; You Can’t Ignore It; You Can’t Fake It) (Morgan & Claypool) 2009. My MS thesis at Michigan was on the design of multi-level cache hierarchies; my Ph.D. thesis at Michigan was on the topic of virtual memory system architectures. I have written over eighty refereed articles on memory systems, cache systems, computer design, embedded systems, operating systems design, astrophysics, and algorithmic composition. These can be found listed in my *curriculum vitae*. As of Winter 2024/2025, my body of research has been cited over 8500 times, with ~85 cites per work, on average. I hold three patents in the area of circuit design and another two the area of memory-systems design.

(7) I founded the annual *International Symposium on Memory Systems* and am frequently invited to give keynote speeches and high-level briefings on the topic of memory systems around the world. I have been the keynote speaker at meetings including EMS, MCC, MCHPC, ModSim, SAMOS, and Computing Frontiers. I was asked to speak before a National Academies panel on the topic of memory systems and to participate in a roundtable discussion with the Department of

Energy (DOE) to create a roadmap for computer science and applied mathematics research at DOE in the coming decades.

(8) Topics my research group has researched extensively include *DRAM and main memory technologies* (e.g., controllers, devices, bus organizations, system architectures, and memory-system power), *non-volatile memories* (e.g., flash memories, numerous resistive RAM variants, and controller & system designs for both main memory and storage), and *embedded systems*. The last area includes investigation of topics in design verification for heterogeneous systems (e.g., the detailed, hardware-verified simulation of heterogeneous systems, for the design and verification of embedded systems that control various hardware devices in real time), highly integrated, heterogeneous systems-on-a-chip (e.g., design and implementation of a monolithic *micro electro mechanical systems* [MEMS]-based gas sensor component, integrated on silicon with a digital microcontroller), and the hardware/software co-design of embedded microcontrollers and real-time operating systems.

(9) In summary, I have extensive technical experience and expertise relating to aspects of designing and implementing computer memories, including the fabrication of semiconductor devices and chips, all of which are the topics of concern in this matter.

(10) My *curriculum vitae* is attached as Exhibit XYZ.

(11) The following is a brief overview of my opinions.

II. LEGAL CONSIDERATIONS

A. Level of Skill in the Art

(12) In my opinion, a person of ordinary skill in the art in the early 2000s (e.g., May 2001 or March 2002) would have had an understanding of multiprocessor design and/or multicore microprocessor design, as well as multiprocessor cache design. The person of ordinary skill would have had a bachelor's degree in computer science, electrical engineering, or computer engineering, and two years of relevant experience in memory-systems design, and related fields. Lack of professional experience can be remedied by additional education, and vice versa. I was a person of more than ordinary skill in the early 2000s.

B. Claim Construction

(13) It is my understanding that in order to properly evaluate the '264 Patent, the terms of the claims must first be interpreted. It is my understanding that for the purposes of this *inter partes* review, the claims are to be construed under the so-called Phillips standard, under which claim terms are given their ordinary and customary meaning as would have been understood by a POSITA in light of the specification and prosecution history, unless the inventor has set forth a special meaning for a term. I have also been informed that claim terms only need to be construed to the extent necessary to resolve the unpatentability inquiry.

(14) I do not believe that any claim constructions would affect my analysis or conclusion that the claims are obvious over the asserted prior art.

C. Identifications of How the Claims Are Unpatentable

(15) I am not aware of any secondary considerations. The Patent Owners have not identified any secondary considerations of nonobviousness so I have not considered any in my analysis.

(16) I have considered the teachings in the prior art from the perspective of a POSITA and it is my conclusion that a POSITA would have found it obvious to combine the teachings of those known elements in the way they are recited in the Challenged Claims as of March 6, 2002, or May, 15, 2001.

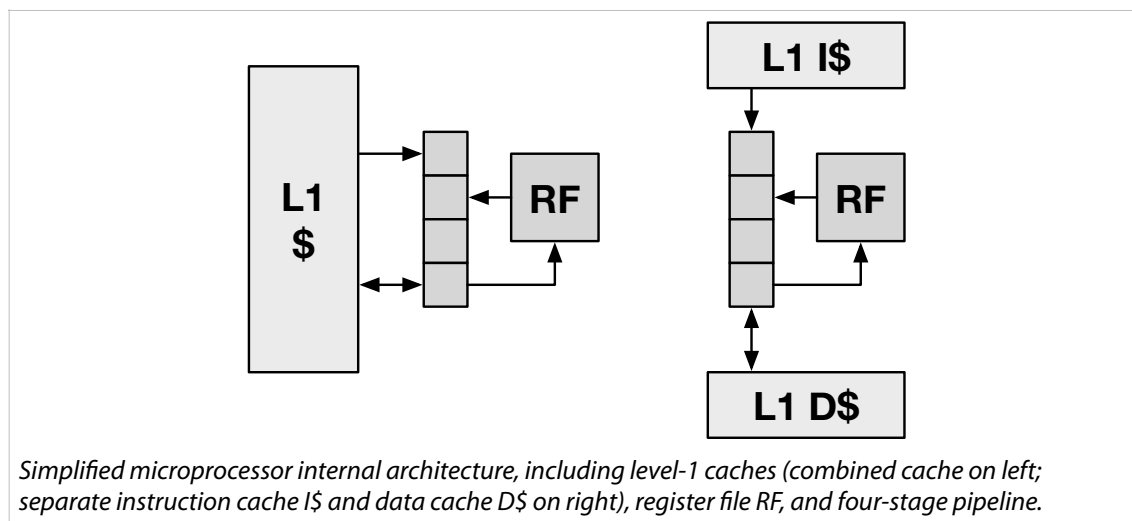
(17) Regardless of whether secondary considerations exist or not, they would not rise to the level to overcome the strong obviousness arguments presented below. I do not believe any secondary considerations would alter my overall conclusion that the claims would have been obvious to a POSITA as of March 6, 2002, or May 15, 2001. Nevertheless, should the Patent Owners provide secondary considerations and argue those considerations rise to the level of maintaining the validity of the Challenged Claims, I reserve the right to respond.

IV. TECHNICAL BACKGROUND

(18) The '264 patent claims are directed to *multiple-instruction-stream processors* (for instance, *multicore* processors) that have *multi-level caches*. The following provides some technical background on this topic. In general, see [Hennessy & Patterson 2011, Jacob et al. 2007, and Sorin et al. 2011].

A. Multicore Processors

(19) While there are many different processor architectures that would fall under the category *multiple-instruction-stream*, by far the most common is multicore. This section provides a background on the topic. The figure below shows a simplified illustration of a microprocessor. It includes a four-stage pipeline (the portion that does the work, in particular the part that executes software instructions and processes data), and several memory components: a register file, RF, and two caches. These latter structures are what store the software instructions and data to be processed.



(20) The caches are simply memory structures that hold instructions and data.

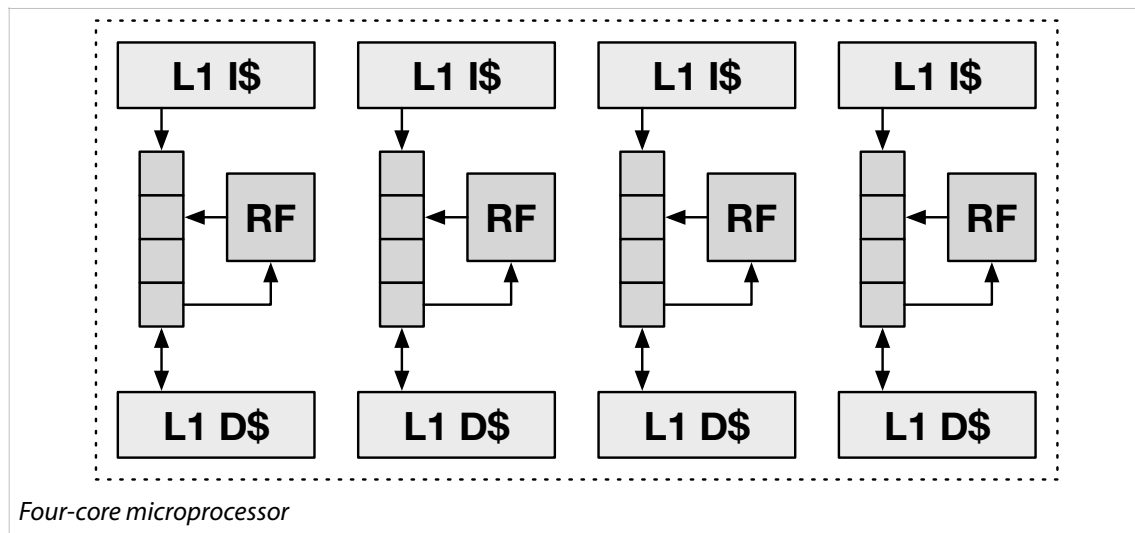
The caches pictured include two types:

- On the left, a combined instruction+data first-level cache labeled “L1 \$” ... this is a unified cache that contains both instructions and data
- On the right, a split first-level cache comprising an instruction cache labeled “L1 I\$” that contains only instructions, and a data cache labeled “L1 D\$” that contains only data and no instructions

(21) The choice of combined cache or split cache is a design option available to the design engineer, and both types of caches were well known to persons of ordinary skill, back into the late 1970s and early 1980s.

(22) The first-level cache feeds the front-end of the pipeline at the top (the front-end of the pipeline is where the instructions are retrieved from memory, and it is at that point the microprocessor begins to interpret and execute the instructions), and the back-end of the pipeline is where all the computed results are saved, so that is where the data is accessed, and that is where the register file is accessed.

(23) In the mid-1990s, some academic researchers proposed to put more than one microprocessor onto a single silicon chip [Olukotun 1996, Hammond 1997]. The concept of a multiprocessor had been well-known for several decades—in a multiprocessor, several microprocessors are combined into a single system, in which they share resources such as main memory, network interface, and disk subsystem. These academic researchers were pointing out that semiconductor manufacturing capabilities at the time were such that one could support the level of integration need to build a full multiprocessor system on a single silicon chip. Such a design looks like the following, in which four microprocessors from the previous figure are combined into a single unit, all implemented on a single silicon chip.



(24) About this time, the terminology came about to call each internal processor a “processor core” or simply “core,” rather than a “microprocessor,” and this terminology is typically used to distinguish a chip-level object from a component within that chip. For instance, using this distinction, one can name something a “multi-core microprocessor” without being unclear.

(25) The following is the proposed chip layout for a four-core multiprocessor, from Olukoton’s 1996 paper:

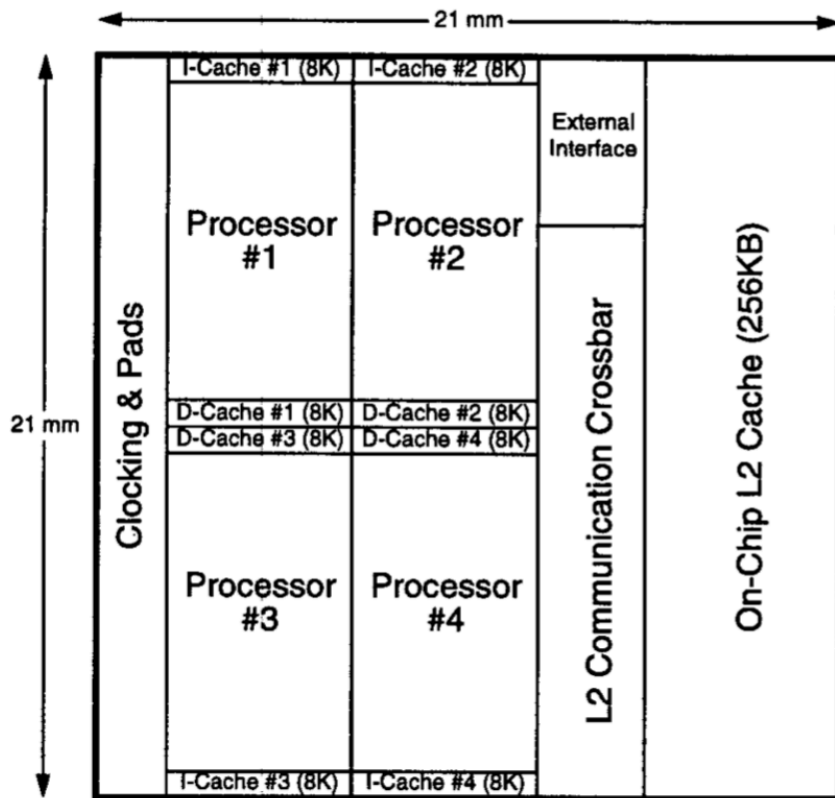
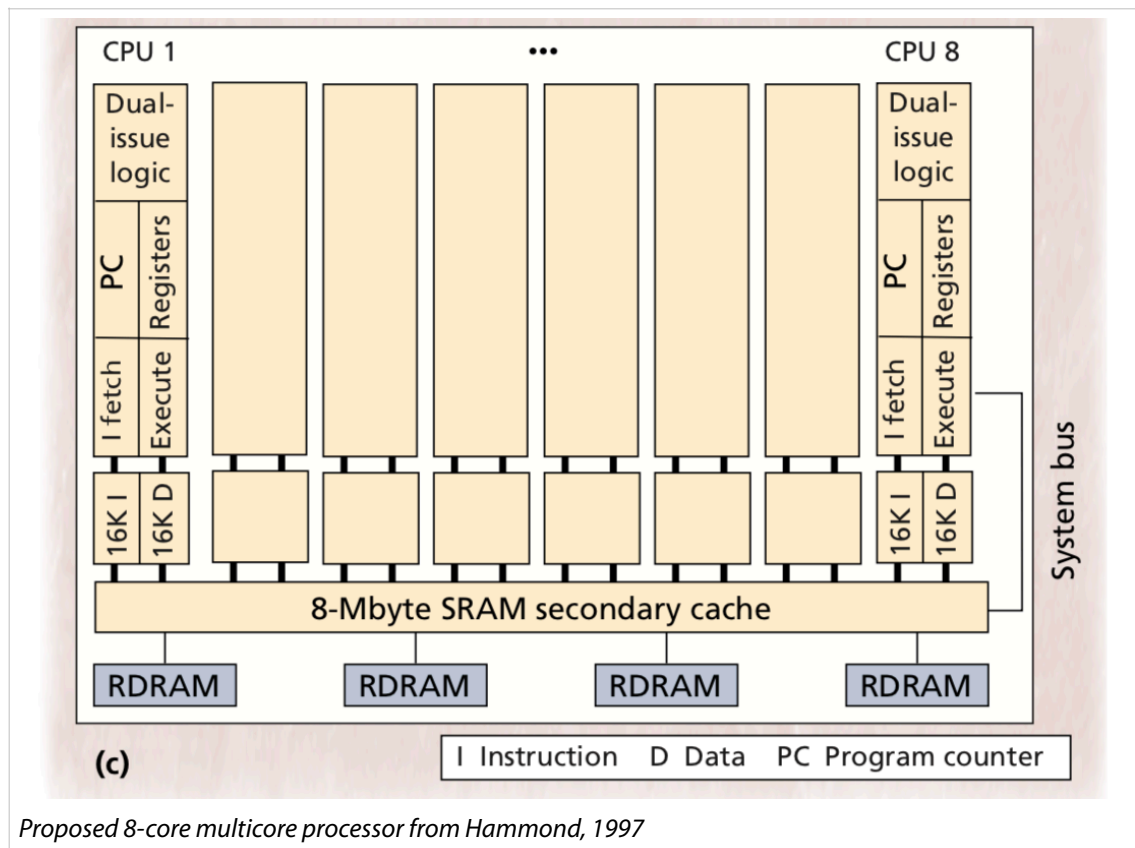


Figure 3. Floorplan for the four-way single-chip multiprocessor.

Proposed 4-core multicore processor from Olukotun, 1996

(26) The design shows four processor cores (Processors #1 through #4) and a *multi-level cache hierarchy* comprising a per-core I-Cache, a per-core D-Cache, and an on-chip shared L2 Cache. The multi-level cache hierarchy is indicated from the fact that each processor core has its own level-one instruction and data caches (the I-Cache and D-Cache components located at the end of each processor), and all four processors share the on-chip level-two cache via a crossbar switch.

(27) Hammond's proposed design shows an eight-core microprocessor with substantially larger caches (e.g., 8MB shared second-level cache vs. the 256KB shared second-level cache from Olukotun 1996):



(28) The following section describes caches and cache hierarchies in more detail.

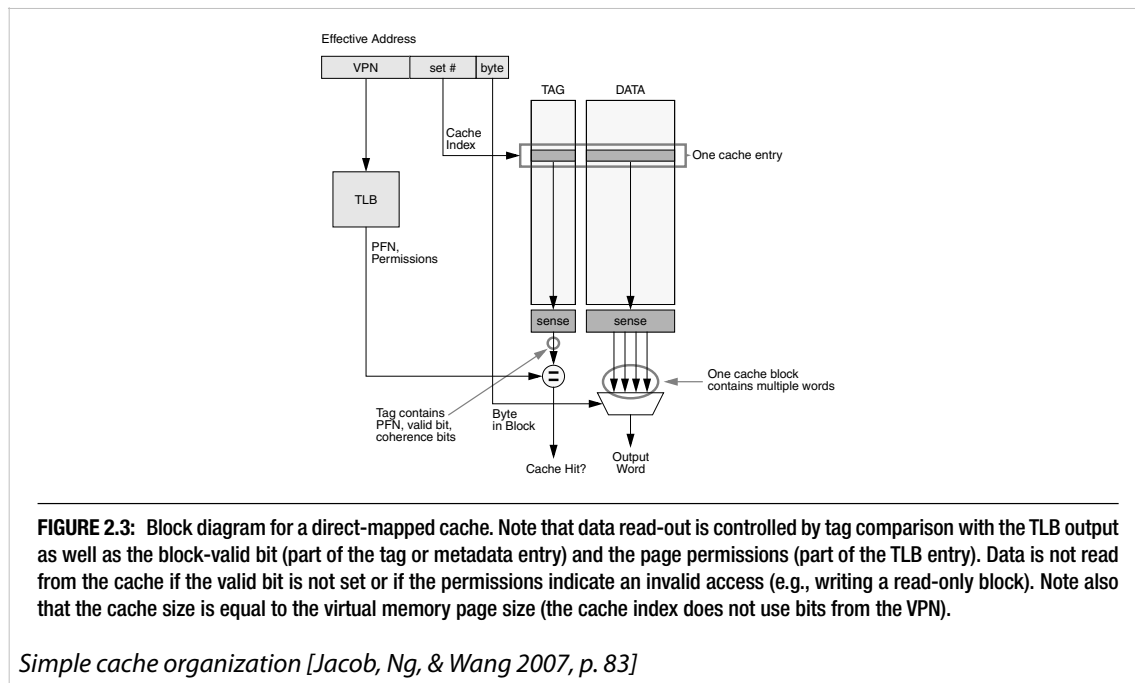
B. Processor Caches and Cache Hierarchies

(29) A cache is a relatively small memory structure that holds copies of data, whether that data is a set of instructions to perform, or the numbers (data) on

which the instructions operate. This leads to caches specialized for holding instructions (“instruction caches” or “i-caches”) and those specialized for holding data (“data caches” or “d-caches”). Caches of both types are connected to processor cores across busses, which are groups of wires that carry the requests and also carry the data involved in those requests. One can see, for example, in the illustration above from Hammond, the dark black lines connecting the various caches to the CPUs, and to the right is an indication that those black wires make up the “system bus” of the single-chip microprocessor. Each of those wires is a bus that carries data between the caches and the processor cores, and each is a separate segment of the larger “system bus” connecting the caches and processor cores of the chip.

1. General Cache Organization and Operation

(30) The figure below presents the organization of a very simple processor cache. Processor caches behave like small hardware databases: a database can store & retrieve *data* tagged with a *key* (e.g., one can use an employee ID to look up an employee’s record in a database: the ID is the key, and the record is the corresponding data); a cache stores and retrieves data in the form of a cache *block* (some number of contiguous bytes from memory) tagged with the *memory address* of the data.



(31) The address is used to search within the cache as shown in the figure: part of the address, the *cache index*, is used to select an entry within the cache; part of the address is used in an equality comparison that verifies whether or not the block already there in the cache is the desired block (if the values match, it is a *cache hit*; otherwise, it is a *cache miss*); and part of the address identifies which byte in the block the request needs. When the required data is found in the cache (a “cache hit”), the need to go to main memory is eliminated, which increases performance significantly, because accessing main memory is time-intensive. Therefore, significant attention (e.g., time, energy, and manpower) is spent on optimizing caches and cache systems so as to maximize the number of caches (the *cache hit rate*

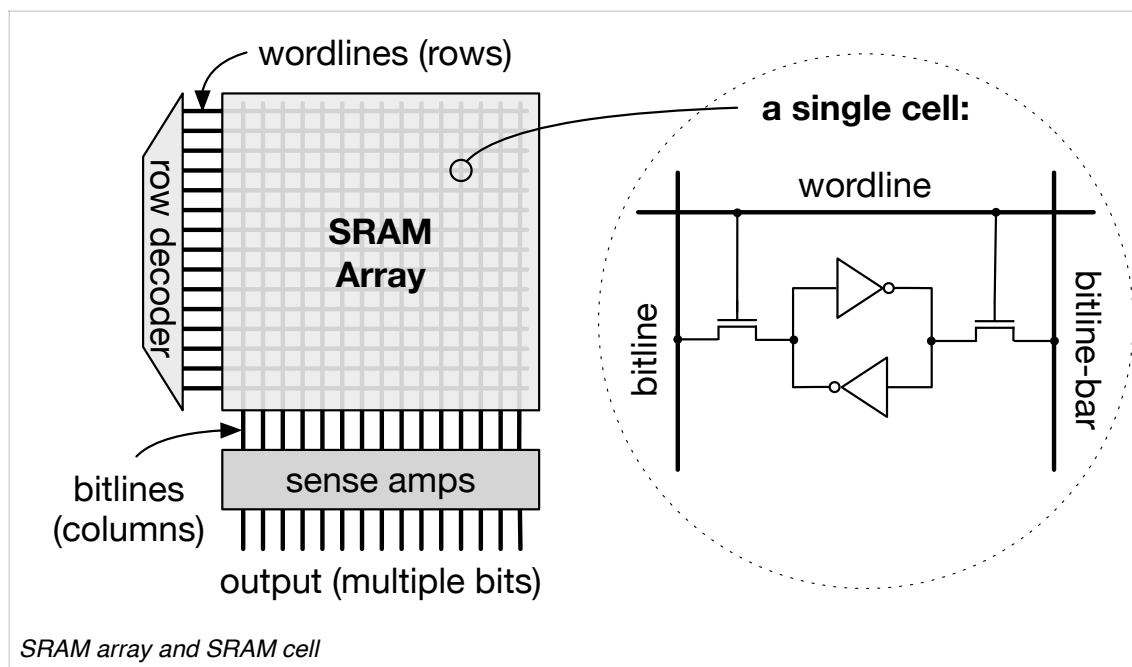
or *hit ratio*), and/or minimize the number of cache misses (the cache *miss rate* or *miss ratio*). See, for example, Alan J. Smith's seminal work on caches [Smith 1982], which provides graphs of cache performance as measurements of cache *miss ratio* for different cache configurations.

(32) Another aspect shown in the diagram is that the information stored in the *tags array* can be more extensive than just a simple key value. Items mentioned in the diagram include the PFN (the Page Frame Number, which is the memory address that serves as the block's key), a valid indicator, and additional *status* indicators such as *cache-coherence* information. The hardware that manages the cache read/write operations is called the *cache controller*. Among other things, a cache controller ensures the correct operation of the local cache and also maintains integrity between the cache and the backing store and maintains coherence in multiprocessor cache systems.

2. Multilevel Caches

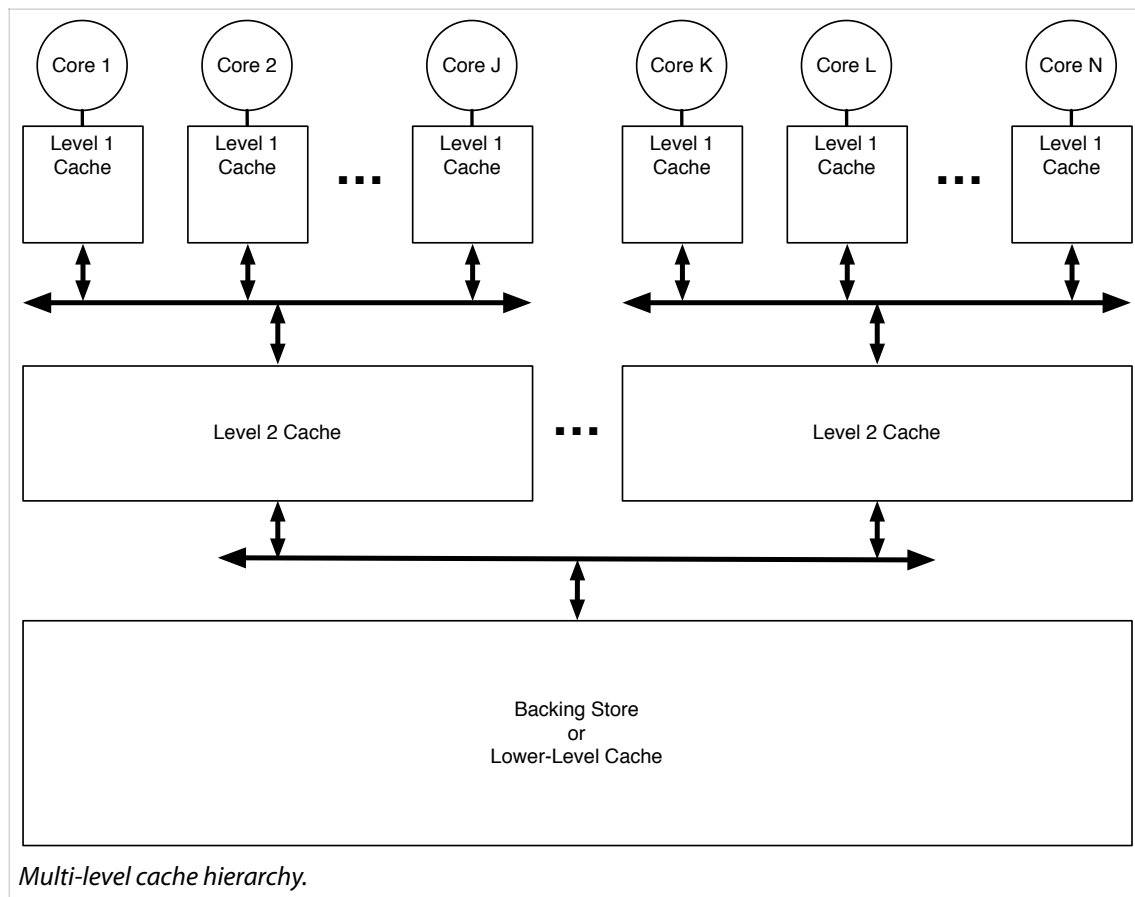
(33) Processor caches are typically made of *static random-access memory (SRAM)*, a semiconductor memory technology. Like most other semiconductor memory technologies, SRAM is built as an array of *wordlines* and *bitlines* (metal wires) arranged in a north-south/east-west configuration like the streets of a city (this is

called a “Manhattan” layout). At each intersection is a storage cell. The figure below illustrates the over-all organization of the array.



(34) The power required to drive this array is proportional to the physical size of the array, and the physical size of the array depends on the capacity of the array. Thus, when very large-capacity caches are desired, it is often more efficient to create a two-level *cache hierarchy*, in which a small (and thus low-power) cache sits in front of a much larger cache. Most of the requests will be satisfied out of the small cache, and so most of the requests can be handled with relatively little power. Only those requests that *miss* the first-level cache will need to access the second-level cache; only those requests will exact the higher power costs inherent in accessing a

larger cache. The following figure illustrates an example multi-level cache hierarchy.



(35) In the example, each processor core has its own dedicated first-level cache, and a sub-group of the processor cores all share a second-level cache. One or more other sub-groups have similar shared-second-level cache arrangements, and beyond those second-level caches is either another level of cache or main memory, often called “backing store.” The use of multi-level cache hierarchies has been a popular

technique in computer-systems design for many decades (see, for example, the seminal paper on the topic by Alan J. Smith [Smith 1982]). In the 1980s, each of the components in the figure above would have been a separate device. In the early 1990s, it was commonplace for the processor core and first-level cache to be integrated on the same chip, but large second-level caches were often off-chip. By the late 1990s, large second-level caches were integrated on chip, and, as shown above, researchers were proposing to integrate multiple processor cores onto the chip as well.

(36) Three of the important issues in multi-level caches are the following:

- ***Shared vs. Private***, indicating which cores have direct access to which caches
- ***Cache coherence***, a hardware mechanism that attempts to synchronize read/write behavior across multiple caches
- ***Inclusion***, the decision (or not) to ensure that lower levels of the cache hierarchy contain everything found in the higher levels

(37) These require special protocols to be implemented on the various busses interconnecting the components, and they are addressed in more detail in the following subsections.

a. Shared Caches vs. Private Caches

(38) Processor caches are either shared among multiple processor cores, or they are private to a single core, meaning that no other processor core has direct access to the cache. The difference is typically illustrated in the bus diagrams indicating the cache organization. For example, the hierarchical-cache figure above shows both private and shared caches.

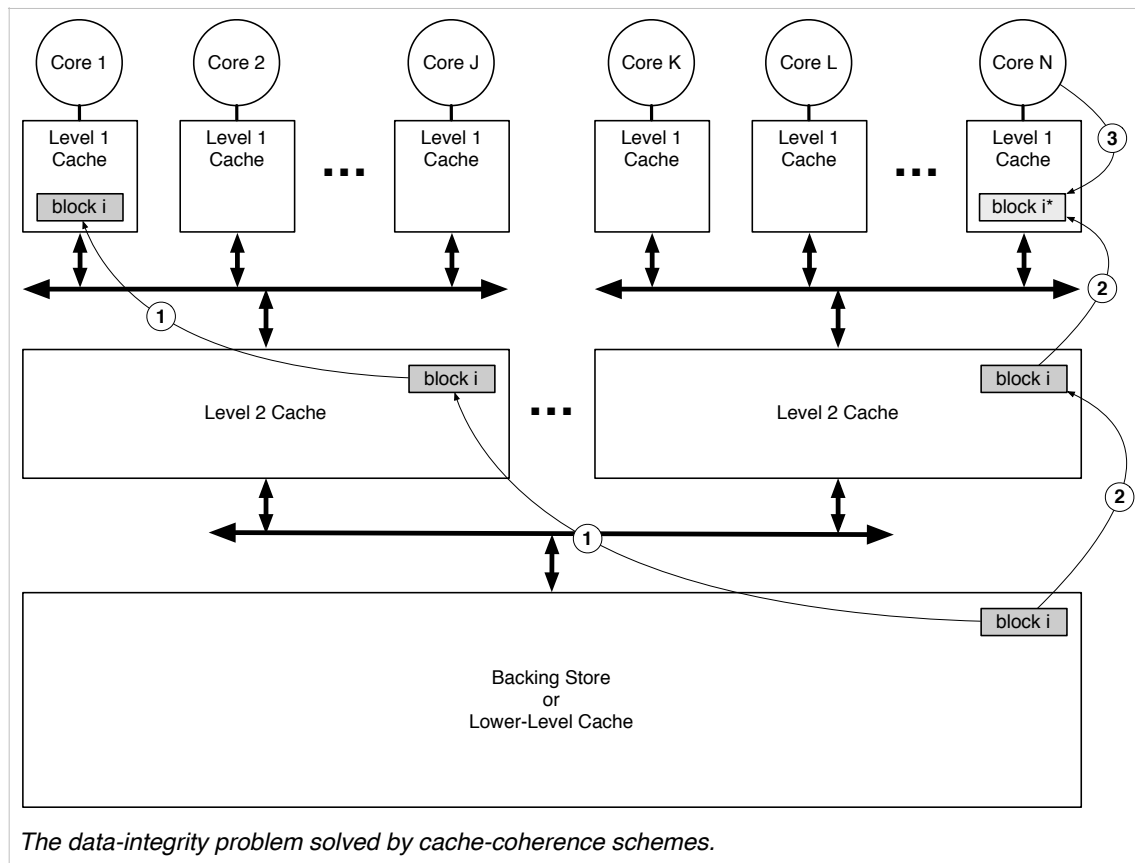
- The Level-1 caches are all shown as being private to their respective cores. This is indicated by the fact that each core is connected directly to its cache, and each cache is connected only to one core. In addition, the figure shows that there is exactly one cache per core, which generally indicates that the caches are private, unless there is reason to believe otherwise.
- The Level-2 caches are all shown as being shared among multiple, but not all, cores. For instance, the Level-2 cache on the left is shared by Cores 1 through J, and the Level-2 cache on the right is shared by Cores K through N. Cores 1 through J have direct access to the Level-2 cache on the left, and Cores K through N have direct access to the Level-2 cache on the right.

- The Level-3 Cache on the bottom (as the figure shows, it can either be backing store or a “lower level cache”) is shown as being shared among all of the cores in the system: Cores 1 through N. All cores have direct access to this cache.

b. Cache Coherence

(39) Cache coherence is designed to prevent problems such as the one illustrated in the following figure. The figure shows the multi-level cache hierarchy from before, where multiple processor cores each have their own first-level cache, and each N-core cluster of processor cores shares a second-level cache. In this diagram, the whole system maintains cache coherence: to implement this, the individual caches on each shared bus watch the transactions on that bus to make sure that they conform to whatever transactions are going on in the system.

(40) In the figure, both Core 1 and Core N have read *block i* into their respective caches (**steps 1 & 2**), and Core N has further overwritten *block i* with a new value (**step 3**), indicated by the asterisk, and also indicated by the block having a lighter shade of grey.



(41) Why this presents a problem may not be immediately obvious.

(42) This scenario is analogous to the following. You are the last-level cache or main memory (the bottommost rectangle in the diagram), and you are coordinating a meet-up at a favorite restaurant with two friends. The first friend, Core 1, sends you a text message asking when the meet time is, and you respond with a text message telling him to meet at 7pm (**step 1**: the *block-i* data is transferred from the main memory to the shared L2 cache, and from there to Core 1's L1 cache). The second friend, Core N, also sends a text asking for the time, and you text back that

it is 7pm (**step 2**: the *block-i* data is transferred from the L2 cache to Core N's L1 cache). However, when the second friend, Core N, receives the text, he realizes that he cannot make it until 8pm. At this point, he has effectively overwritten the shared information with a new meet-up time (**step 3**: the data is overwritten in Core N's cache). However, at this point in time, nobody else knows it: not you, and not Core 1.

(43) The problem with the dinner arrangements will not be solved unless your friend Core N tells everyone that the meeting time has to be changed to 8pm. If this information is not conveyed to all parties, then some will arrive at dinner at the wrong time.

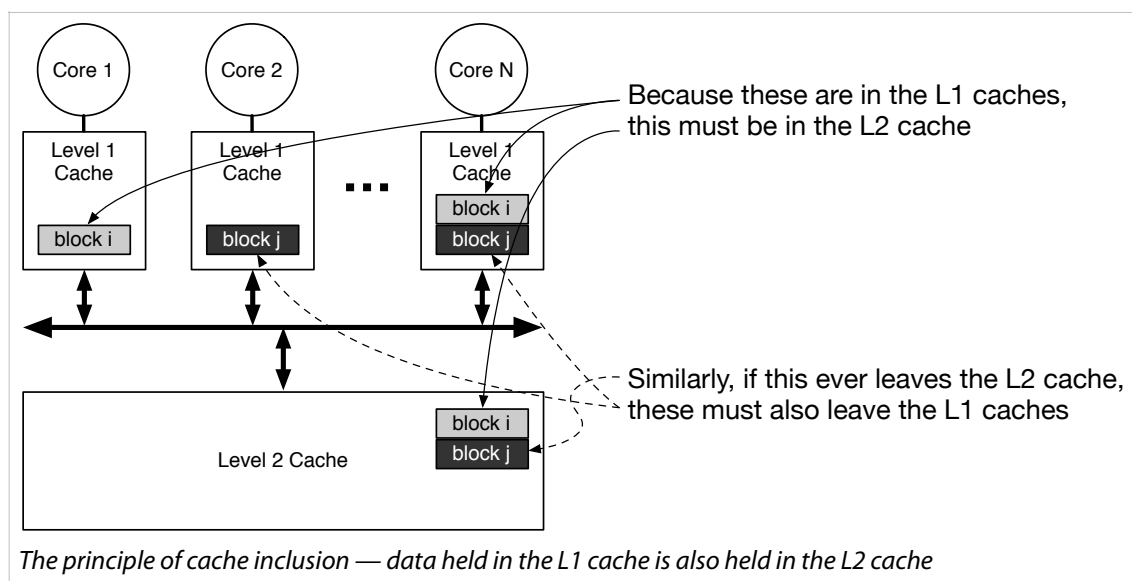
(44) Similarly, the computer-system scenario will cause data-inconsistency errors unless Core N communicates its updated value of *block i* to the rest of the system. At the point in time represented by the figure, Core 1, both second-level caches, and the backing store all have out-of-date copies of *block i*. This is referred to as having “stale” data. If Core 1 uses its copy of *block i*, then it will produce a computed result that is based on old, out-of-date, stale data. This could very well cause erroneous behavior, potentially on a system-wide level. The stale copy of *block i* in the second-level caches and backing store is also a problem, because if any

other processor core asks for the data in that cache block, the processor core will obtain stale information, and the same situation as with Core 1 will arise.

(45) A processor system's *cache coherence* mechanism addresses this problem. The mechanism keeps state on each cache block in the system, and when a processor core tries to overwrite a block, the processor core must first broadcast to the entire system that it is about to write that block, so that other caches can first invalidate their copies. This prevents the scenario depicted above where two different caches at the same level can have different versions of the same data. Cache coherence has been in wide use since the 1980s.

*c. **Cache Inclusion***

(46) Cache inclusion is a design principle that, when enforced, ensures that any data in the Level-1 cache is also held in the Level-2 cache; i.e., the Level-2 cache contains a superset of the data held in all the Level-1 caches. Similarly for the L2-vs-L3, and so on. The principle is illustrated in the following figure.

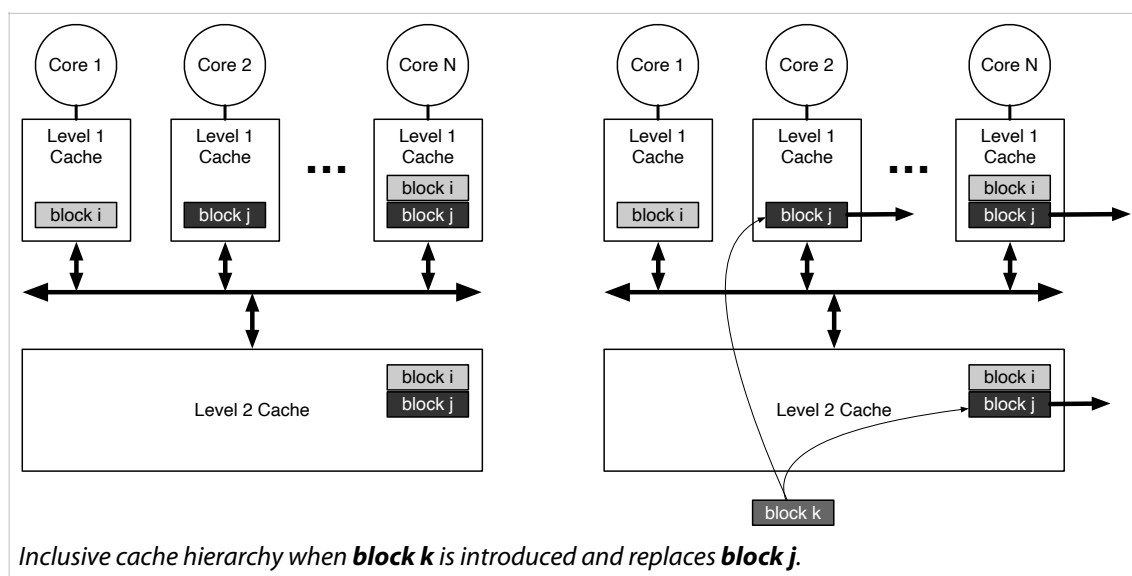


(47) The choice of whether to maintain inclusion impacts the cache-hierarchy implementation significantly. The following is a brief discussion of the issues to consider in the choice of *inclusive* versus *exclusive* or *non-inclusive* caches.

(48) **Inclusive caches** simplify the coherence mechanism in many ways; for instance, if a cache hierarchy implements inclusion, then the coherence scheme need probe only the lowest level of a client's cache when performing invalidates or updates: if a block is not found in the lowest level of an inclusive cache hierarchy, then the block cannot be found at a higher level. This allows a coherence probe to proceed without disturbing the core's interaction with the L1 cache. However, the duplication of data (and wastage of die area) in an inclusive cache hierarchy can

become significant, especially if multiple processor cores and their associated level-1 caches all feed off the same shared level-2 cache.

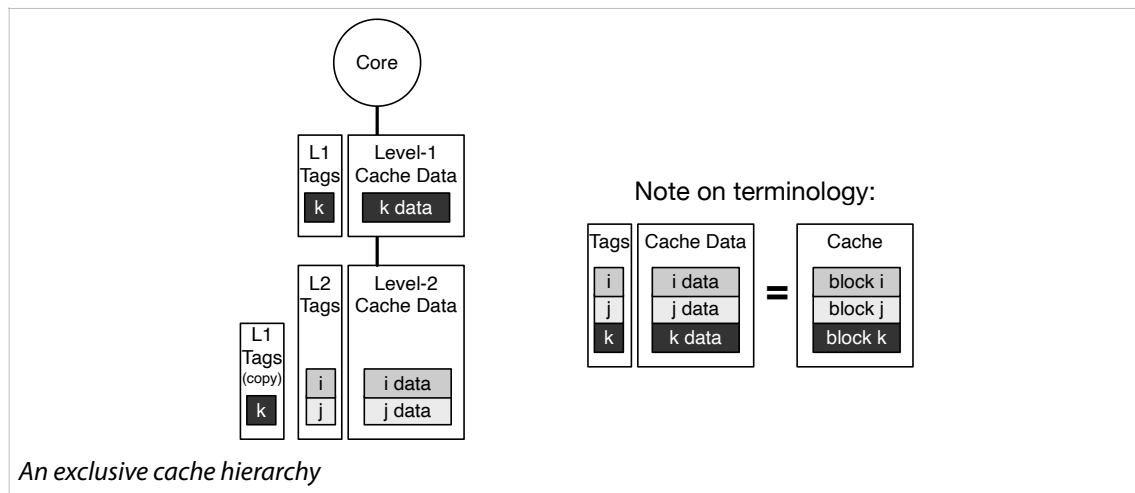
(49) The following diagram shows an inclusive cache in action, illustrating the behavior raised by the previous figure.



(50) In this example, when **block k** is brought into the L1 cache of Core 2, there is a *cache conflict*, meaning that the cache organization is such that the cache cannot hold both **block j** and **block k** at the same time, and neither can the L2 cache. One of the blocks must go, and the cache hierarchy chooses to keep the newest block, so **block j** is thrown out. This means that **block j** must be thrown out of the L2 cache as well. This has ripple effects: with **block j** removed from the L2 cache, the block cannot appear in *any* L1 cache above that L2 cache, and so the block

must be removed from the L1 cache of Core N as well. To maintain inclusion in such a scenario, one must probe every single L1 cache in the system (all L1 caches feeding into that L2 cache), and flush all copies of **block j** found, to ensure that inclusion is maintained and no data inconsistencies can occur. This is considered an expensive operation, and the cost is proportional to the number of cores and L1 caches that share the same L2 cache. The coherence engine and cache controllers make all of this happen.

(51) The advantage of **exclusive caches** is that they avoid the duplication of data and wastage of die area associated with inclusive caches. A particular block can either be in the Level-1 cache or the Level-2 cache, but it cannot be in both. While this saves die area and increases available cache capacity (there is no overlap in content between the two caches), the costs are significant: a coherence probe must search the entire hierarchy, not just the lowest-level cache, potentially interrupting/stalling each processor core to probe its L1 cache. In practice, however, rather than probing all levels of the cache hierarchy, the tagsets are typically replicated down at the last-level cache to centralize the information needed for a coherence check. Thus, any status held in the L1 cache is also held in the L2 cache. The following figure gives a brief look at this arrangement:



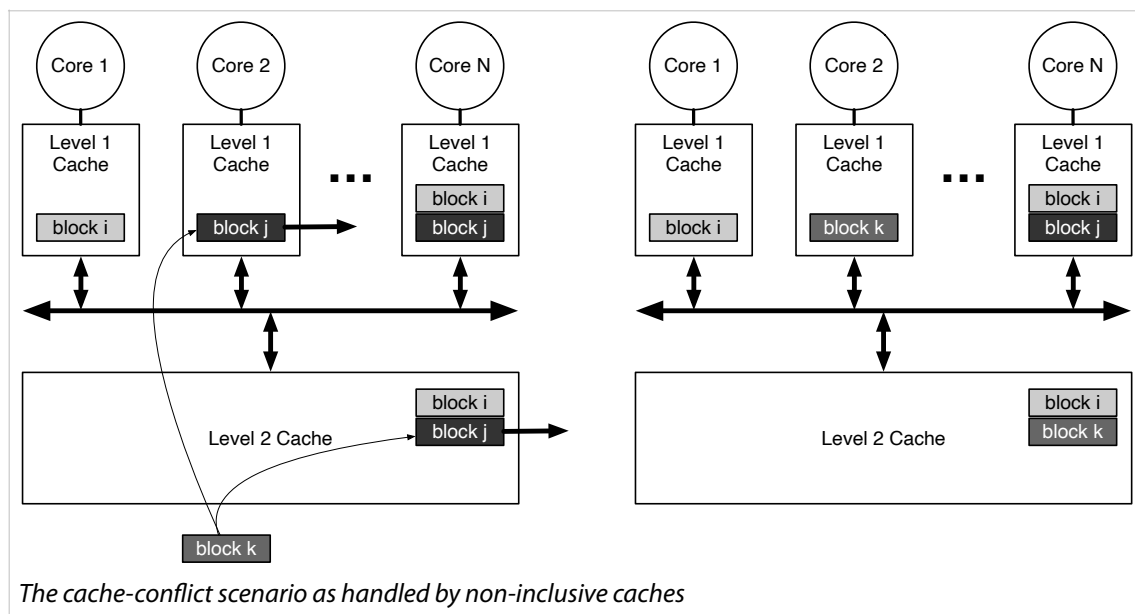
(52) Note that this duplication of L1 tags is something that is not found in an inclusive cache hierarchy and so represents a die-area increase.

(53) **Non-inclusive caches** attempt to achieve the best of both worlds. In particular, they address a rare but important scenario in which inclusive caches are forced to do something inefficient, which was illustrated in the previous figure showing the ripple effects of removing a block from an L1 cache, and how it can effect all other L1 caches in the system. Inclusive caches absolutely must maintain the inclusion principle, but when there are many L1 caches, it is often difficult to ensure that the inclusion principle holds across the entire hierarchy, and when done so the management aspects can prove to be expensive in both time and energy.

(54) To solve this problem in a more efficient way, *non-inclusive* caches were invented. A non-inclusive cache tries neither to ensure exclusion nor to maintain

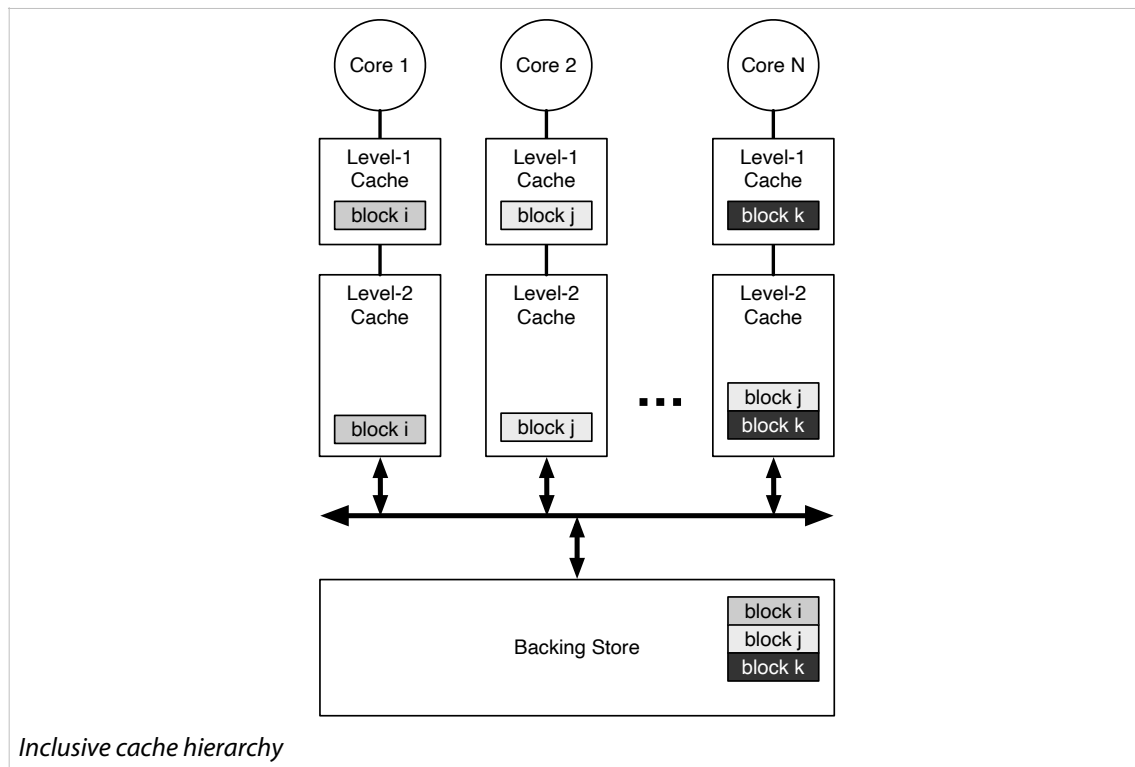
inclusion. An *exclusive* cache will ensure that no data from a block promoted from the L2 cache to the L1 cache remains in the L2 cache; in contrast, a *non-inclusive* cache makes no attempt to ensure this. Similarly, an *inclusive* cache must have hardware to detect and to prevent the scenario illustrated above, whereas a *non-inclusive* cache will allow it. The disadvantage of the scheme is that, like the exclusive cache hierarchy, the scheme must maintain copies of the tags and coherence status across all cache levels.

(55) The following shows the same scenario as above, but it indicates how the non-inclusive cache would handle the scenario:

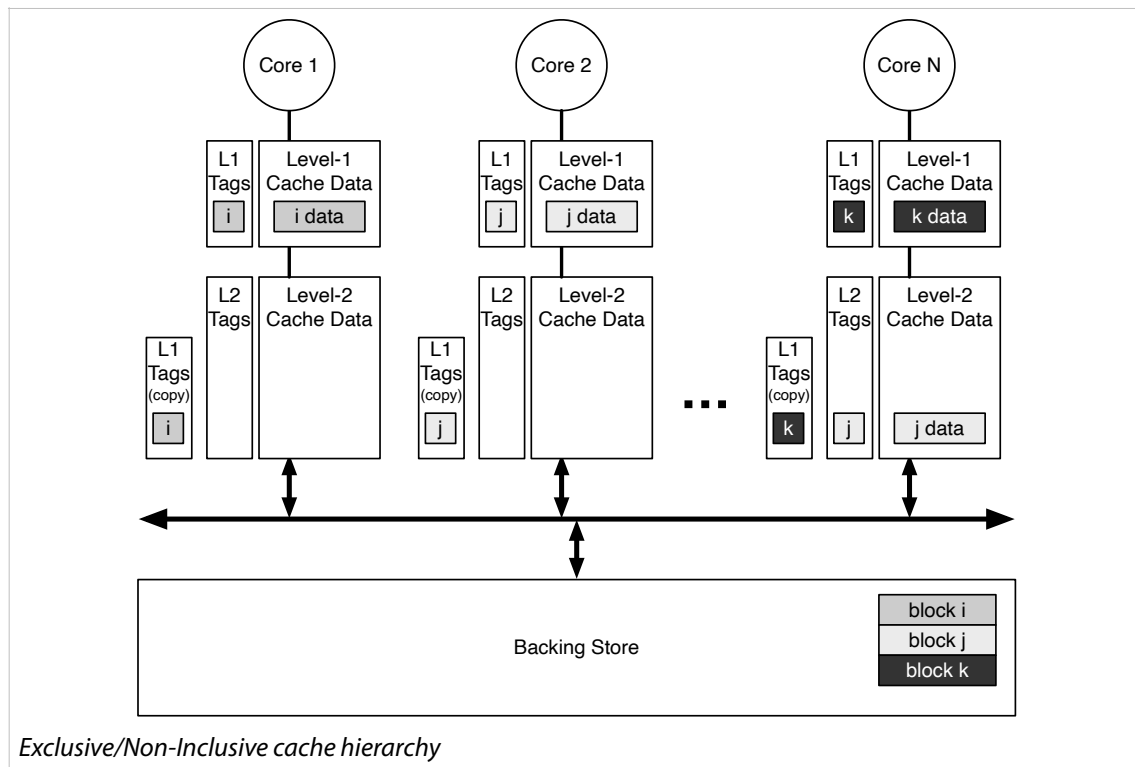


(56) Here, the cache conflict in Core 2's L1 cache is present, as well as the conflict in the shared L2 cache, and **block j** must be removed from Core 2's cache as well as the shared L2 cache. However, no other replacement is needed. In particular, **block j** need not be removed from Core N's cache at all, nor any of the caches between Core 2 and Core N. Like the exclusive caches, the coherence engine requires that the entire cache hierarchy be probed on coherence events, which is made possible by replicating the L1 tagsets down at the L2 cache level.

(57) The following figures illustrate the differences between the different cache organizations. The first is how an inclusive cache would handle a set of data items. Note that the "backing store" is either an L3 cache or main memory.



(58) The next figure shows the same data set in either an exclusive or non-inclusive cache hierarchy, noting the use of replicated tags at the lower cache levels.



(59) In this example, the backing store can only be main memory for the exclusive example (the arrangement of data blocks in the figure above would violate the principle of exclusion if it were an L3 cache), but it can be either an L3 cache or main memory if this is a non-inclusive cache hierarchy.

(60) As mentioned above, the coherence engine and cache controllers monitor the bus transactions, probe the appropriate tagsets, and respond however appropriately (e.g., by invalidating cache blocks that would otherwise cause inconsistency), and by doing so they maintain cache coherence and therefore data integrity and consistency across the system.

V. OVERVIEW OF U.S. PATENT 6,871,264

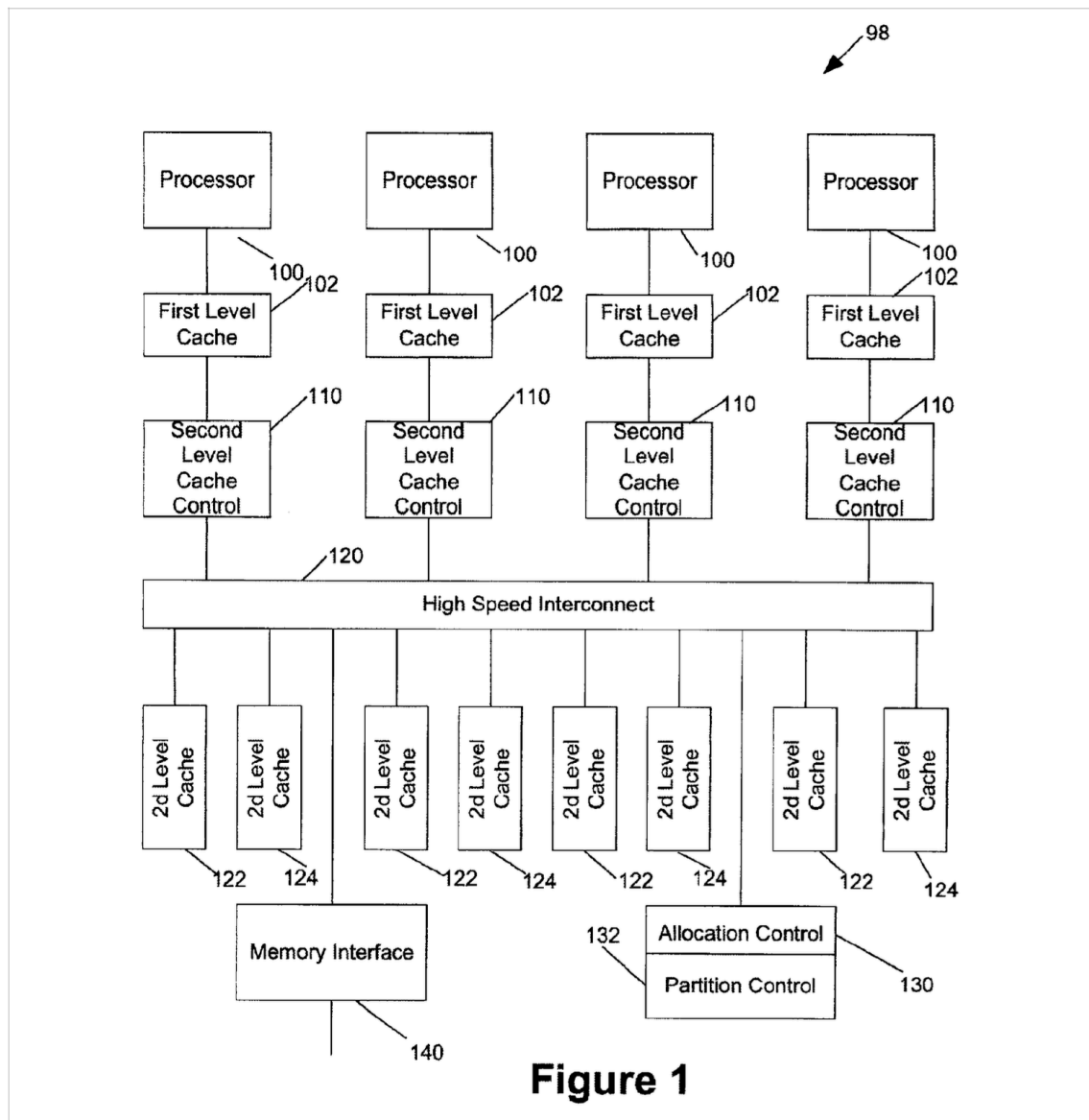
A. Patent Disclosure

(61) U.S. Patent 6,871,264 (“the ’264 patent”) describes a multiprocessor system in which the processors, coupled together, share a level of cache memory in a way that portions of the cache memory can be assigned to certain processors and therefore be dedicated to them.

(62) The patent’s Abstract is as follows:

A processor integrated circuit capable of executing more than one instruction stream has two or more processors. Each processor accesses instructions and data through a cache controller. There are multiple blocks of cache memory. Some blocks of cache memory may optionally be directly attached to particular cache controllers. The cache controllers access at least some of the multiple blocks of cache memory through high speed interconnect, these blocks being dynamically allocable to more than one cache controller. A resource allocation controller determines which cache memory controller has access to the dynamically allocable cache memory block. In an embodiment the cache controllers and cache memory blocks are associated with second level cache, each processor accesses the second level cache controllers upon missing in a first level cache of fixed size. [’264 patent, *Abstract*]

(63) The patent's Figure 1 illustrates the system, including the various processors and the shared cache blocks (the set of blocks labeled "2nd Level Cache" and numbered 122, 124) that can be dynamically allocated to different processors:



(64) The '264 patent describes this as follows:

A partitionable multiple-processor integrated circuit 98 (FIG. 1) has several processors 100. Each processor performs memory references for instructions data through a first level cache 102. Those references that miss in first level cache are directed to second level cache controllers 110.

Each second level cache controller 110 is coupled to a high-speed interconnect 120. High-speed interconnect 120 allows the second level cache controllers 110 to each access one or more of a plurality of cache memory blocks 122 and 124. Of these memory blocks, at least one is an allocable cache memory block 124 that may be allocated to any cache controller 110. There may, but need not, be one or more cache memory blocks 122 for which allocation is fixed. Allocation controller 130 determines which, if any, of the dynamically allocated cache memory blocks 124 is accessed by each cache controller 110. Partition control 132 operates to determine which processors are associated with each system partition. Partition control 132 and allocation controller 130 therefore together determine the second level cache controller 110, processor 100, and system partition each allocable cache memory block 122 and 124 is associated with. [’264 patent, 4:9–31]

(65) The patent’s Figure 2 shows more details of the shared cache’s design and operation:

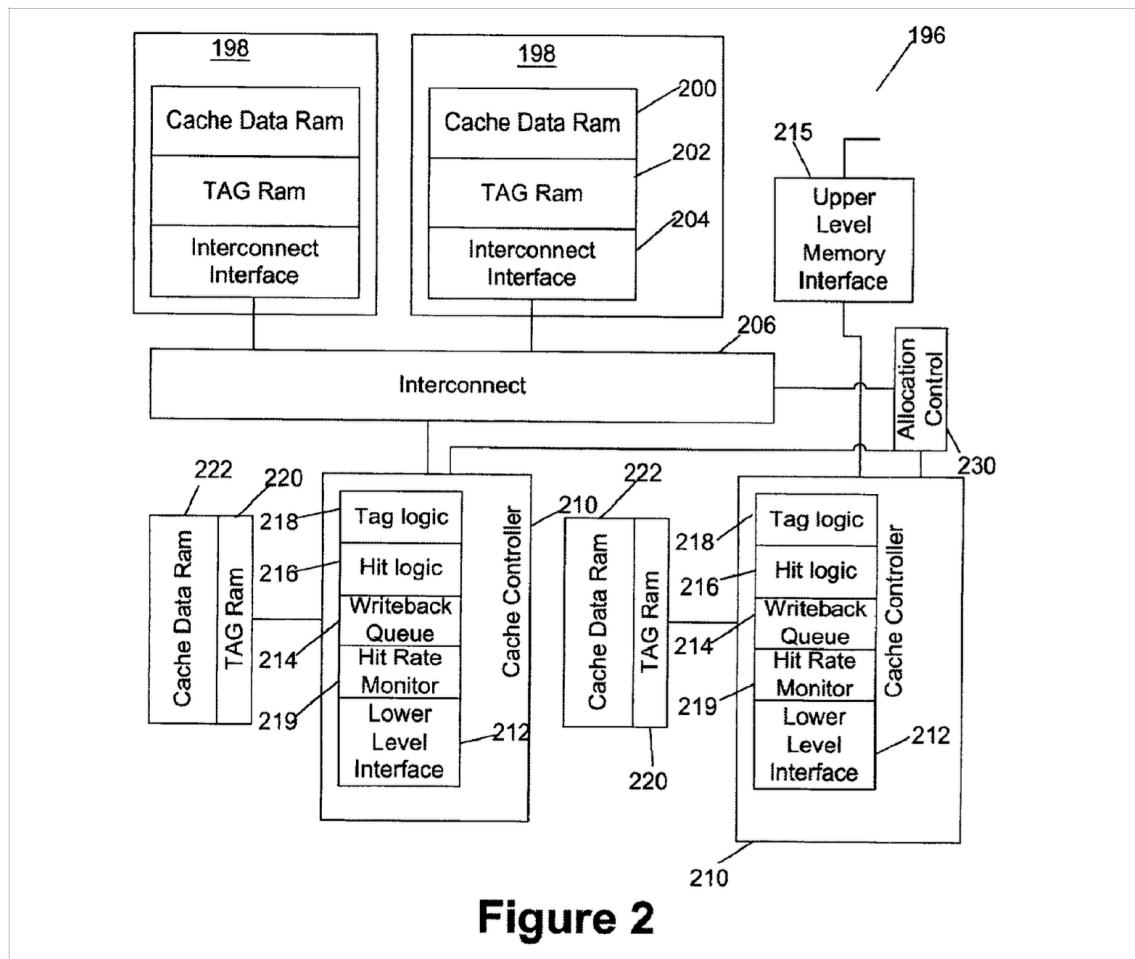


Figure 2

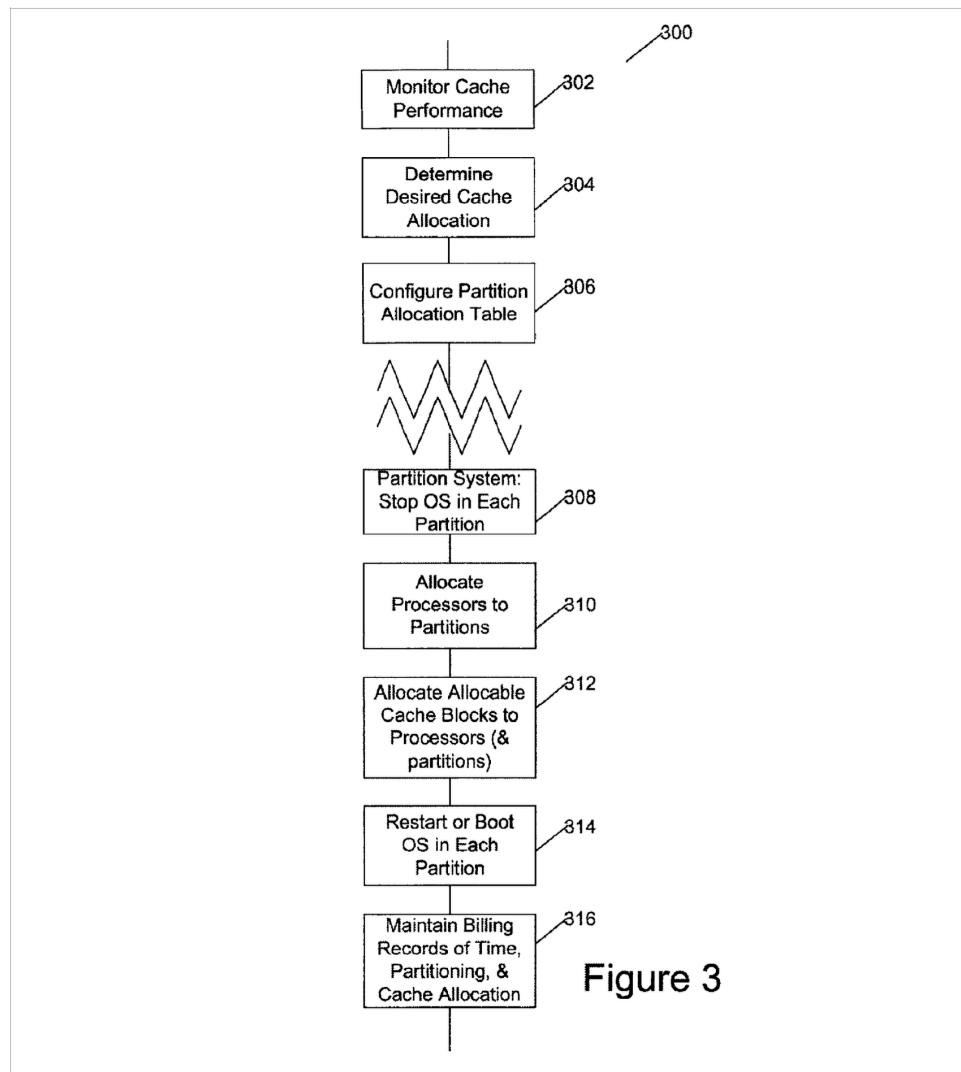
(66) The patent describes this as follows:

In an alternative embodiment of the integrated circuit 196, each dynamically allocated cache memory block 198 (FIG. 2) incorporates data memory 200, tag memory 202, and an interface 204 to interconnect 206. Each cache controller 210 has an interface 212 for receiving memory references that miss in lower level cache. Each cache controller 210 also has cache control logic as known in the art, including a writeback queue 214 for tracking cache writes and

controlling writeback to main memory or higher level cache through upper level memory interface 215, as well as tag logic 218 and hit logic 216 for determining those references that are located in cache. Each cache controller also incorporates hit rate monitor logic 219 for measuring cache performance including hit rate. The cache controller 210 has local tag memory 220 and data memory 222. Tag memory 220 and data memory 222 are sufficiently large that they can provide a high hit rate for some instruction streams that have small cache requirements.

Cache controllers 210 connect to interconnect 206, which allows them to communicate with dynamically allocated cache memory blocks 198 under control of allocation control 230. Allocation control 230 and interconnect 206 is configured by software to connect zero, one, or more dynamically allocated cache memory blocks 198 to each cache controller 210. Each dynamically allocated cache memory block 198 may only be accessed by only one cache controller 210 at any given time. ['264 patent, 4:36–62]

(67) The system is very similar to a traditional multiprocessor with shared cache (as described in the *Background* section), with some of the claimed inventions including the use of a cache hit rate monitor to determine how best to allocate the blocks among the processors. This is shown in Figure 3 and its associated discussion:



A method 300 (FIG. 3) for managing a computer system built around one or more of the multiprocessor integrated circuits 98 (FIG. 1), includes monitoring 302 (FIG. 3) cache performance. Monitoring makes use of the hit rate monitor 219 shown on FIG. 2, and such other cache performance monitor apparatus as may be incorporated into the integrated circuits, to make a record of past cache performance of partitions, applications, and prior cache allocation. This record of past performance under various conditions is used to determine 304 a desired system partitioning taking into account cache

availability on the one or more multiprocessor integrated circuits 98.

In a particular embodiment, the volume of past interprocessor communications is also considered in determining which processors to place in each system partition. The determined partitioning is configured 306 into a partition allocation table. [’264 patent, 5:1–16]

(68) Additionally, there is a claim regarding whether or not the operating system needs rebooting when a partition ... the only description of such a mechanism is shown in the figure above and the following paragraph:

The system is repartitioned each time it is booted, and is also capable of being dynamically repartitioned at other times when repartitioning is appropriate to improve overall system performance. Repartitioning requires that any running operating system in each affected partition be stopped 308. The processors 100 of each integrated circuit 98 are assigned 310 to partitions according to the partition allocation table. Then, the dynamically allocable cache blocks 124 are assigned 312 to processors 100 of each partition according to the partition allocation table. Next, the operating systems for each partition are booted, or restarted, 314; and billing records are maintained 316 of machine time, system partitioning, and cache allocation. These billing records permit charging customers according to the number of processors and amount of cache assigned to their applications. [’264 patent, 5:17–32]

B. The Claims of the '264 Patent

(69) The following are those Claims of the '264 patent that are being challenged in the associated Petition and analyzed in this Declaration:

1. A processor integrated circuit capable of executing more than one instruction stream comprising: a first processor, coupled to fetch instructions and access data through a first cache controller; a second processor, coupled to fetch instructions and access data through a second cache controller; a plurality of cache memory blocks; a high-speed interconnect coupling the plurality of cache memory blocks to the first and second cache controllers such that at least one allocable cache memory block is capable of being used by the first and second cache controllers; and a resource allocation controller coupled to determine an accessing cache memory controller selected from the group consisting of the first and second cache memory controllers, whereby the accessing cache memory controller is allowed to access the allocable cache memory block, wherein the cache memory blocks are usable by the cache controllers to store data and instructions fetched from a random-access memory.
2. The processor integrated circuit of claim 1, further comprising a plurality of first level cache systems, wherein the first processor fetches instructions and accesses data from the first cache controller through a first first level cache system, and wherein the second processor fetches instructions and accesses data from the second cache controller through a second first level cache system.
3. The processor integrated circuit of claim 1, wherein the cache memory blocks further comprise cache tag memory.
4. The processor integrated circuit of claim 1, wherein each cache controller is provided with cache hit rate monitoring apparatus.

5. A method of dynamically allocating cache on a multiple-processor integrated circuit, where the multiple processor integrated circuit is used in a partitionable multiple-processor system and comprises:

a plurality of processors each coupled to receive instructions from a first level cache associated therewith,

a plurality of allocable upper level cache memory blocks, interconnect apparatus for transmitting cache misses at each first level cache to upper level cache memory blocks assigned thereto, and

allocation apparatus for assigning upper level cache memory blocks to processors;

the method comprising the steps of:

monitoring past cache performance associated with processors and partitions;

determining desired processor to partition and upper level cache block allocations to processors; and

repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits,

wherein the upper level cache blocks are usable to store data and instructions fetched from a random-access main memory.

8. The method of claim 5, wherein the multiple processor integrated circuit further comprises a plurality of non-allocable cache memory blocks.

9. The method of claim 5, wherein the interconnect apparatus further comprises a plurality of upper level cache controllers, and where each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache.

10. The method of claim 5, wherein each upper level allocable cache block further comprises tag memory and cache data memory.

11. The method of claim 10, further comprising the steps of stopping execution of operating systems in each partition, and restarting execution of operating systems in each partition, and wherein the system is capable of being repartitioned without rebooting each operating system.

VI. OVERVIEW OF THE PRIOR ART

A. United States Patent 6,725,336 by Rajasekhar Cherabuddi (“Cherabuddi”)

(70) United States Patent 6,725,336 (“Cherabuddi”) describes a multiprocessor system in which the processors, coupled together, share a level of cache memory in a way that portions of the cache memory can be assigned to certain processors and therefore be dedicated to them. Cherabuddi’s Abstract reads as follows:

The resources of a partitioned cache memory are dynamically allocated between two or more processors on a multiprocessor unit (MPU). In one embodiment, the MPU includes first and second processors, and the cache memory includes first and second partitions. A cache access circuit selectively transfers data between the cache memory partitions to maximize cache resources. In one mode, both processors are active and may simultaneously execute separate instruction threads. In this mode, the cache access circuit allocates the first cache memory partition as dedicated cache memory for the first processor, and allocates the second cache memory partition as dedicated cache memory for the second processor. In another mode, one processor is active, and the other processor is inactive. In this mode, the cache access circuit allocates both the first and second cache memory partitions as cache memory for the active processor.

[Cherabuddi, *Abstract*]

(71) The system is shown in the patent’s Figure 2 and its associated description:

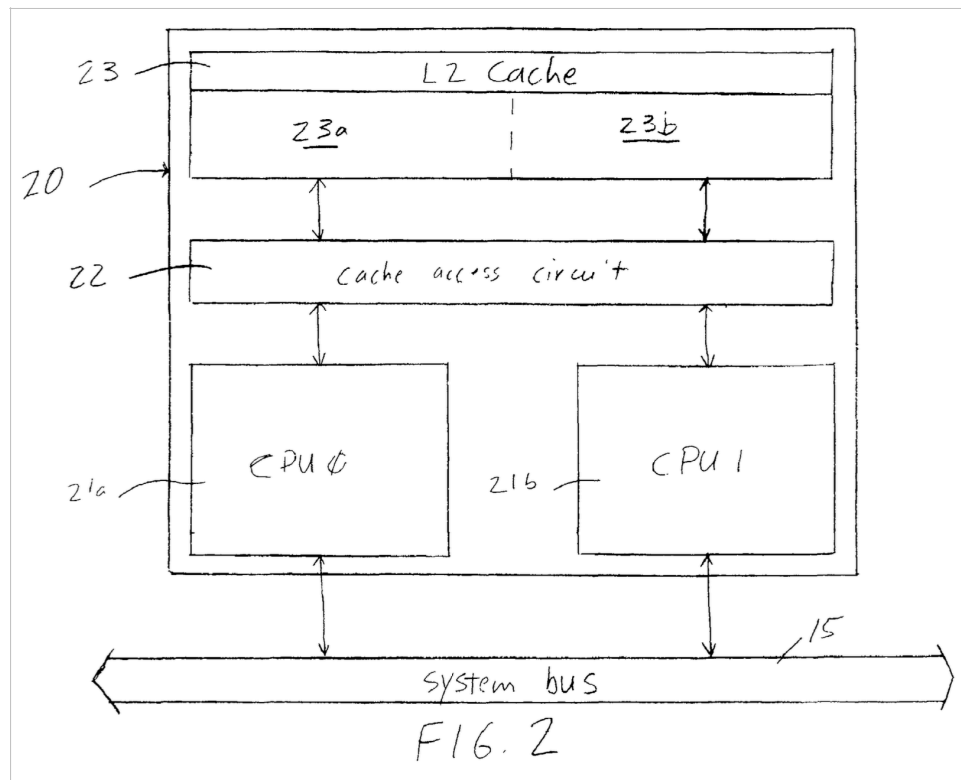


FIG. 2 shows an MPU 20 that is one embodiment of MPU 11 of FIG. 1. MPU 20 is shown to include first and second processors such as central processing units (CPUs) 21 a-21 b, a cache access circuit 22, and a dynamically allocated cache memory 23. CPUs 21 a-21 b are well-known processing devices. Cache memory 23 is partitioned into first and second cache memory partitions 23 a-23 b, and is preferably a high speed cache memory device such as SRAM, although other cache devices may be used. For the purpose of discussion herein, each cache memory partition 23 a-23 b is configured as a 2-way associative cache memory. Of course, in actual embodiments, the cache memory partitions may be configured for other levels of associativity.

Cache access circuit 22 selectively couples the first and/or second CPUs 21 a-21 b to the first and/or second cache memory partitions 23 a-23 b. As explained in detail below, cache access circuit 22 allows

the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21 a-21 b according to each CPU's processing requirements to more efficiently utilize cache resources.
[Cherabuddi, 3:26–46]

(72) Cherabuddi's Figure 4 shows the details of the cache mechanism ... it is presented below with its associated discussion:

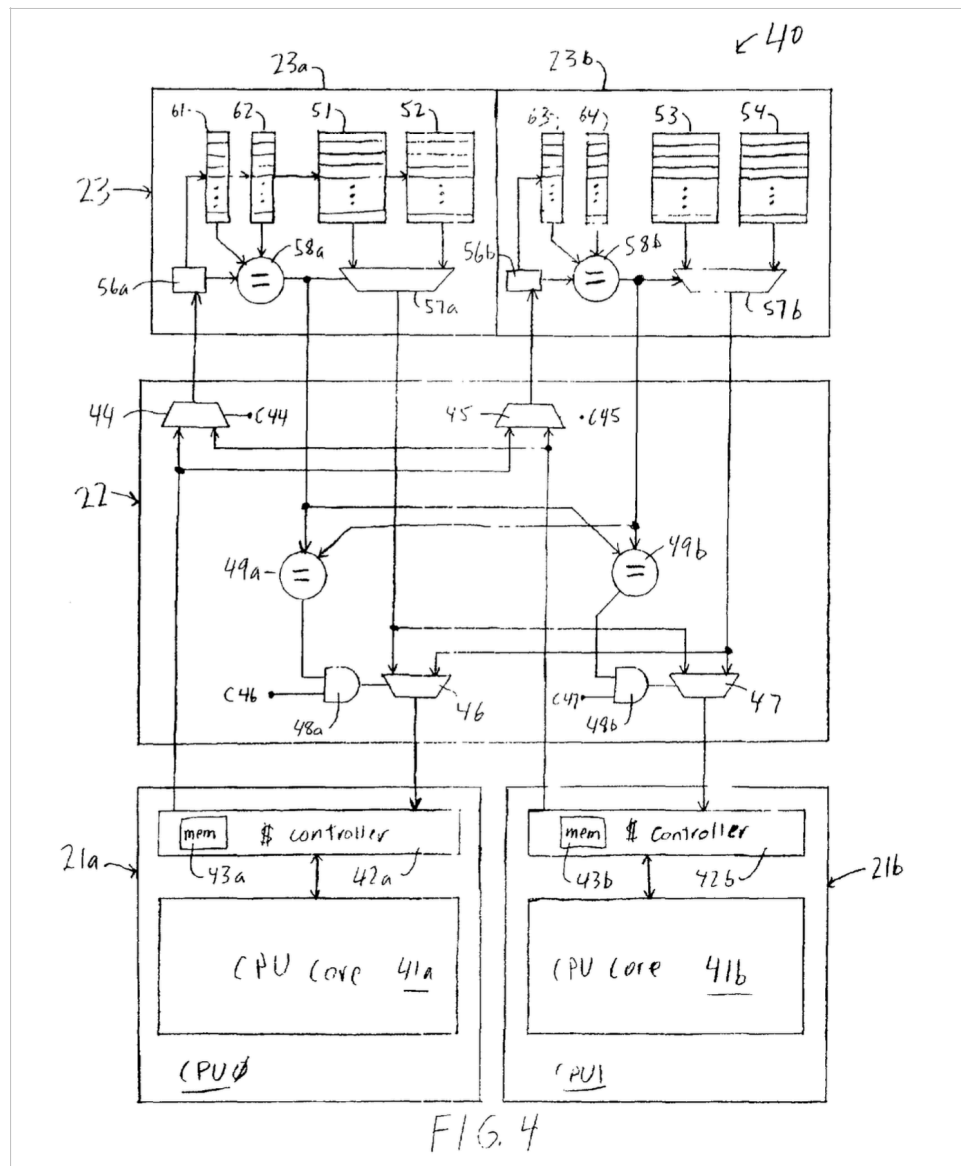


FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23 a-23 b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be

any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.

Cache memory partition 23 a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23 b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23 a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23 b includes a well-known address converter 56 b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54. [Cherabuddi, 4:66–5:28]

(73) Cherabuddi describes in some detail the operation of the cache system, which allows the cache memory blocks (23a and 23b) to be partitioned to serve either CPU Core 41a or CPU Core 41b or both. The mechanism is described by Cherabuddi as being in either a one-thread (1T) or two-thread (2T) state. In the 1T mode, one thread is running on one CPU core, and that threads gets the entirety of the cache to itself; in the 2T mode, both CPU cores are active, and they shared the

cache blocks. The mechanism is described below, with reference to Figure 3 as well as the components of Figures 2 and 4 above.

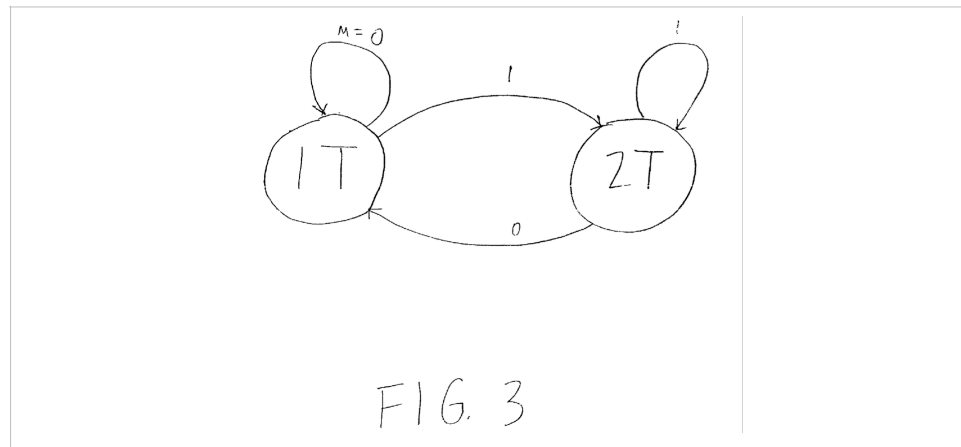


FIG. 3 illustrates state transitions of MPU 20 between a one-thread (1T) state and a two-thread (2T) state. In one embodiment, upon power-up of MPU 20, the kernel sets a mode signal $M=0$ to initialize MPU 20 to the 1T state. The kernel sets one of the CPUs 21 to an active state and sets the other CPU 21 to an inactive state. For purposes of discussion herein, during the 1T state, the kernel sets CPU 21 a as the active CPU and sets CPU 21 b as the inactive CPU, although in other embodiments the kernel may set CPU 21 b as the active CPU and set CPU 21 a as the inactive CPU. While in the 1T state, the kernel assigns tasks of the computer program(s) only to the active CPU 21 a, while the other CPU 21 b remains idle. In response to $M=0$, cache access circuit 22 couples the first CPU 21 a to both the first and second cache memory partitions 23 a-23 b to allow the first CPU 21 a to use all resources of cache memory 23. In this state, the active CPU 21 a may use cache memory partitions 23 a-23 b as a 4-way associative cache memory.

If, during execution of the computer program(s), the kernel

determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21 a-21 b to the active state, and thereafter assigns one execution thread to CPU 21 a and another execution thread to CPU 21 b in a well-known manner. In response to M=1, dirty data in cache memory partition 23 b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23 b. The cache access circuit 22 couples the first CPU 21 a to the first cache memory partition 23 a for exclusive access thereto, and couples the second CPU 21 b to the second cache memory partition 23 b for exclusive access thereto. In this state, CPU 21 a may use cache memory partition 23 a as a dedicated 2-way associative cache memory, and CPU 21 b may use cache memory partition 23 b as a dedicated 2-way associative cache memory.

Thereafter, if the kernel determines that only one of CPUs 21 a-21 b is necessary for a particular instruction code sequence, the kernel may transition MPU 20 to the 1T state by changing the mode signal to M=0, flushing the second cache memory partition 23 b, and then assigning execution of the instruction code sequence to the active CPU 21 a. [Cherabuddi, 3:62–4:37]

(74) Cherabuddi describes the benefits of the system as helping to allocate resources best to the CPU cores involved in a computation, as well as providing a degree of reliability in the case of some portion of the cache being defective:

By dynamically allocating resources of cache memory 23 in response to specific needs of associated CPUs 21 a-21 b, embodiments of the present invention maximize cache performance by ensuring that both cache memory partitions 23 a-23 b are utilized, irrespective of whether one or both CPUs 21a-21 b are active. Thus, in the 1T state, both cache memory partitions 23 a-23 b are allocated to the active CPU, and in the 2T state, each cache memory partition 23 a and 23 b is allocated only to its corresponding CPU 21 a and 21 b, respectively. Since allocation of cache memory partitions 23 a-23 b is controlled by cache access circuit 22, cache memory 23 does not require any special hardware, and thus may be of conventional architecture. Further, since cache memory 23 is not shared between CPUs 21 a-21 b, all snoop operations may be performed at the system level. As a result, the cache controllers (not shown in FIG. 2) in CPUs 21 a-21 b are much simpler and occupy less silicon area than cache controllers for shared cache memory systems.

The ability to dynamically allocate cache resources is also useful in situations where portions of MPU 20 are defective. For example, during testing of MPU 20, if CPU 21 b is found to be defective or otherwise unusable, the kernel may be configured to maintain MPU 20 in the 1T state, where CPU 21 a is the active CPU and has access to both cache memory partitions 23 a-23 b, and CPU 21 b is inactive. Thus, in contrast to MPUs that have dedicated cache memory for each on-board CPU, the failure of one CPU 21 on MPU 20 does not render any part of cache memory 23 inaccessible. [Cherabuddi, 4:38–65]

B. United States Patent 6,604,174 to Alvar Dean, et al. (“Dean”)

(75) United States Patent 6,604,174 (“Dean”) describes a multiprocessor system in which the processors, coupled together, share a level of cache memory in a way that portions of the cache memory can be assigned to certain processors and therefore be dedicated to them. Dean’s Abstract reads as follows:

The present invention provides a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and multiple single-cycle cache accesses from a multiport cache. Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways, but other metrics may also be used.

[Dean, *Abstract*]

(76) Dean describes the general point of the invention and its use and then provides details of the system in the patent’s Figure 1 and its associated description:

The preferred embodiments of the present invention provide a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and

multiple single-cycle cache accesses from a multiport cache. Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways, but other system metrics may also be used.

The system and method preferably determines which process could benefit the most if additional cache space was allocated to the process or could benefit a computer system the most if additional cache space was allocated to the process. The system and method then allocates additional ways to the process, at the expense of removing ways from other processes. The reallocation can take place periodically, thereby maximizing the performance of the system and handling unforeseen demands by processes. Because the multiport tag can look up multiple requests during one cycle, there is reduced waiting for cache access and look up by processes. Moreover, because of the multiported nature of this system and method, multiple accesses by processes are allowed during single cycles. This reduces cache access time and improves overall system performance.

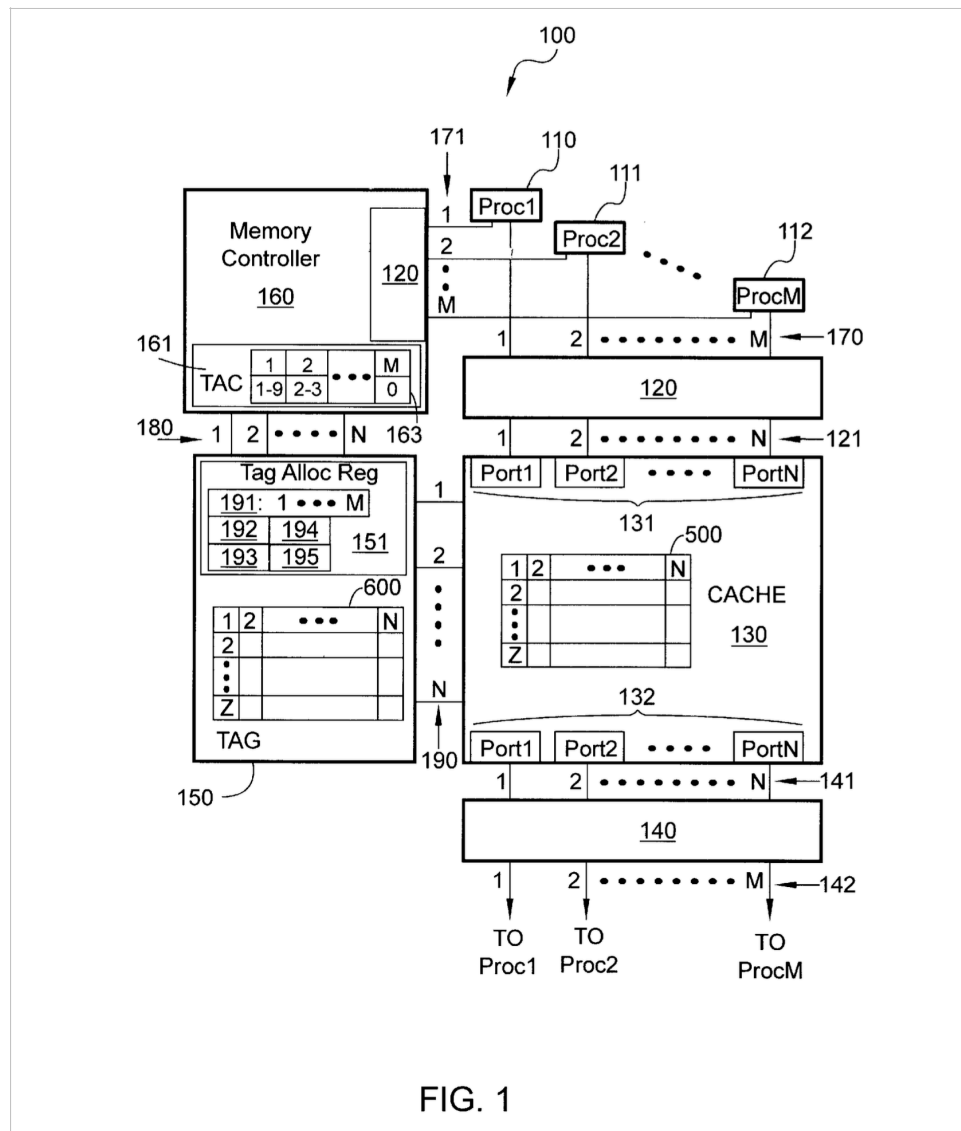


FIG. 1

Referring now to FIG. 1, a system 100 is shown that provides a multiport cache and tag structure and allows dynamic allocation of cache ways. System 100 comprises M processes 110, 111, 112, a processes-to-cache multiplexor 120, a cache-to-processes multiplexor 140, a multiport cache 130, a memory controller 160, and a tag 150. Memory controller 160 comprises a processes-to-cache multiplexor 120, N output addresses 180, and a tag allocation controller 161. Processes-to-cache multiplexor 120 in memory controller 160 has N

outputs (not shown) that are routed to the N addresses 180. Processes 110, 111, 112 can be individual processors that access the memory controller and cache through buses 170, 171 or can be tasks that are running on one processor and accessing the memory controller and cache through buses 170, 171. These processes communicate the address of the data to the memory controller through bus 171, write data through bus 170, and read data through bus 142. [Dean, 3:45–4:23]

(77) Dean says that using the *cache-miss percentage* of each processor is the best way to determine the allocation of memory ... meaning that the system attempts to ensure that all processors are allocated enough space in the cache so that they all achieve roughly the same cache miss rate.

Cache miss percentage 194 is a factor used to determine if the cache should be reallocated. The total misses for all of the process are compared to the cache miss percentage to determine if there are too many misses. The cache miss percentage 194 thus reduces the number of times the cache is reallocated. The cache miss percentage 194 is optional, but preferred. The cache reallocation factor 195 is a factor used to reallocate the cache, and it determines what percentage of the cache will be reallocated to a task. This factor will generally be a certain number of ways of the cache or a certain percentage of the cache. [Dean, 3:45–4:23]

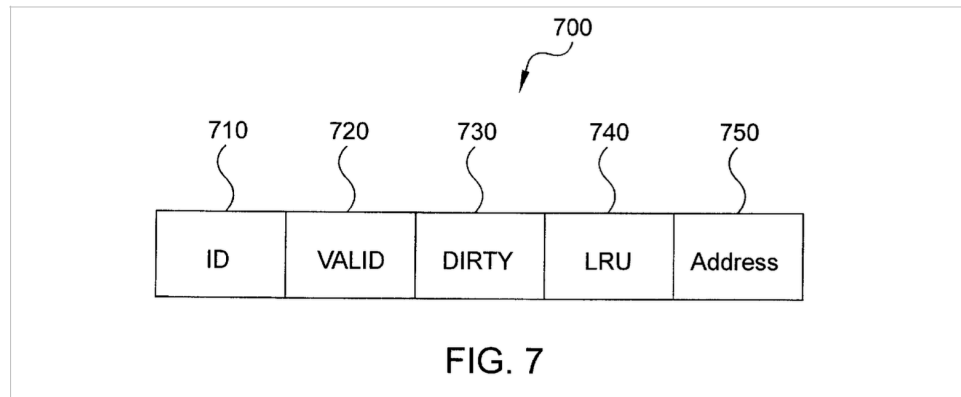
When the cache evaluation period has completed (step 255=YES), it is first determined if the total number of cache misses is large enough to

warrant a cache reallocation cycle (step 260). This is done by adding miss counters A, B, C, and D in system metric 191 and comparing the result to a cache miss percentage 195. The cache miss percentage 195 can be calculated by multiplying the expected or desired cache hit ratio for each process by the cache evaluation period and adding up the results. If the total number of misses is less than the cache miss percentage, this indicates that the current cache allocation is working and the current cache allocation should be left alone. In this event (step 260=NO), the counters are reset for the current cache cycle counter 192 and the system metric counter(s) 191 (step 265), and another cache evaluation period is started (step 220).

If the sum of the cache miss counters in system metric 191 is greater than the cache miss percentage (step 260=YES), step 270 is performed. Method 270 is shown in FIG. 4, and begins when it is decided if the difference between process cache miss counters is enough to warrant a reallocation cycle (step 271).

If, for example, there are a lot of cache misses, uniformly distributed among all the processes, then the current allocation is considered fair and no reallocation is done. Preferably, the cache is only reallocated when there is a relatively significant disparity in cache miss ratios between processes. Because of this, each cache miss counter for each processor in system metric 191 is compared to the others (step 22). If, for example, process A's miss counter is larger than processor B's miss counter by a pre-determined cache reallocation factor 195, some ways of the cache will be assigned to processor A (step 273). Preferably, this is done by determining which process has the largest percentage of the cache and taking cache ways from this process (step 273). [Dean, 9:27-42]

(78) Dean shows that, to dedicate a particular cache block to processor, the ID of that processor is attached to the “tags” portion of the cache block:



An exemplary tag entry 700 is shown in FIG. 7. For each way in cache memory 500, there will preferably be one entry 700. Entry 700 comprises a process identification (ID) 710, a valid indication 720, a dirty indication 730, a Least Recently Used (LRU) indication 740, and a tag address 750. The ID 710 contains data that indicates which of the M processes is assigned to this particular way and this line. Once a particular process is assigned a way of a line, no other process can write to that way of that line. Another process can read from the way of the line, if the line contains data for this other process. Moreover, the process ID 710 allows the tag 150 to determine if a particular way of a line in the cache memory 500 belongs to a process. The valid indication 720 is generally a bit, and is used to determine whether the way of the line is valid. The dirty indication 730 is also usually a bit, and it is used to mark the way of the line as “dirty”, meaning that the cache contains a more recent version of the data. This version should be written back to main memory in accordance with cache coherence protocols.

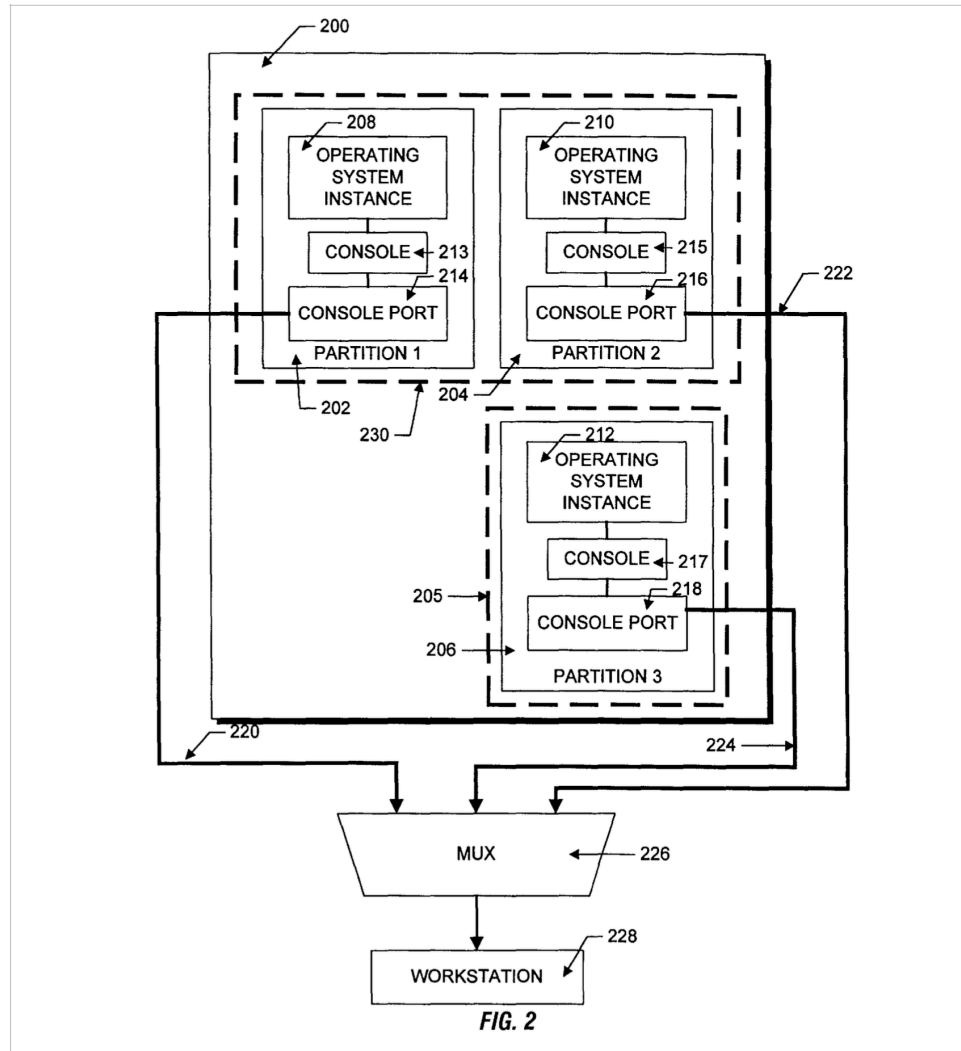
C. United States Patent 6,381,682 to Karen Noel, et al. (“Noel”)

(79) United States Patent 6,381,682 (“Noel”) describes a multiprocessor system in which the processors, coupled together, share a level of memory in a way that portions of the memory can be assigned to certain processors and therefore be dedicated to them. Noel’s Abstract reads as follows:

Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. At different times, different operating system instances may be loaded on a given partition. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. A grouping of partitions, a community, shares memory. Memory may be private to a particular partition or may be shared by partitions within a community. When a community is formed the creating instance reads a configuration tree and builds management structures for the shared memory owned by the community. A single system may have one or more communities, each with their own representation within the configuration tree.

[Noel, *Abstract*]

(80) The system is shown in the patent's Figure 2 and associated discussion:



An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in FIG. 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.

In a preferred embodiment, this allocation is performed by a software program called a “console” program, which, as will

hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in FIG. 2 as programs 213, 215 and 217. The console program may be a modification of an existing administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtualize the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in FIG. 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and 212 to use the resources appropriately and provide coordination of resource allocation and sharing. The hardware platform may optionally provide hardware assistance for the division of resources, and may provide fault barriers to minimize the ability of an operating system to corrupt memory, or affect devices controlled by another operating system copy.

The execution environment for a single copy of an operating system, such as copy 208 is called a “partition” 202, and the executing operating system 208 in partition 202 is called “instance” 208. Each operating system instance is capable of booting and running independently of all other operating system instances in the computer system, and can cooperatively take part in sharing resources between operating system instances as described below. [Noel, 6:49–7:19]

(81) As described in the passage above, Noel divides the system into “partitions,” each of which has an instance of an operating system running in it. In the passage

below, Noel describes how the system's memory resources can be divided between the different partitions. Noel explains that, in some instances, the operating system instances need to be rebooted at the time of repartitioning, and in other instances, the operating system instances ***do not*** need to be rebooted.

Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot. [Noel, 22:8–27]

VII. ANALYSIS OF THE CLAIMS OF U.S. PATENT 6,871,264

(82) In my opinion, the claims of the '264 patent would have been rendered obvious by the prior art. The following sections present the specific combinations and the reasoning for my opinions.

A. Claims 1–3, 5, and 8–10 Are Rendered Obvious by Cherabuddi

(83) In my opinion, Cherabuddi renders obvious Claims 1–3, 5, and 8–10 of the '264 patent. My analysis, which includes my reasoning and the bases for my opinion, is provided below.

1. Claim 1

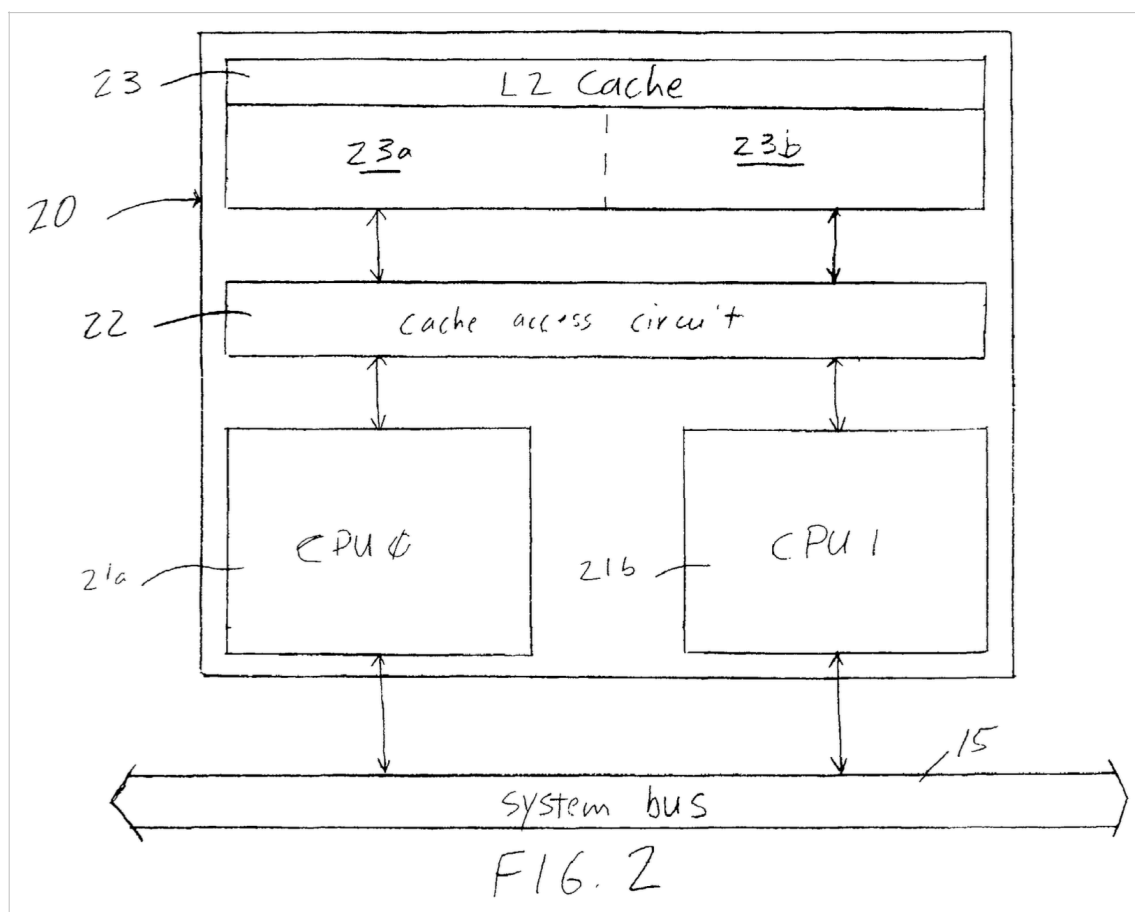
(84) Claim 1 reads as follows:

1. A processor integrated circuit capable of executing more than one instruction stream comprising:
a first processor, coupled to fetch instructions and access data through a first cache controller;
a second processor, coupled to fetch instructions and access data through a second cache controller;
a plurality of cache memory blocks;
a high-speed interconnect coupling the plurality of cache memory blocks to the first and second cache controllers such that at least one allocable cache memory block is capable of being used by the first and second cache controllers; and
a resource allocation controller coupled to determine an accessing cache memory controller selected from the group consisting of the first and second cache memory controllers, whereby the accessing cache memory controller is allowed to access the allocable cache memory block,
wherein the cache memory blocks are usable by the cache controllers to store data and instructions fetched from a random-access memory.

(85) The following sections analyze this claim in light of the prior art.

a) A processor integrated circuit capable of executing more than one instruction stream comprising:

(86) Cherabuddi discloses a single-chip multiprocessor system, illustrated in the figure below:



(87) Cherabuddi describes the system explicitly as a single-chip with multiple central processing units (CPUs):

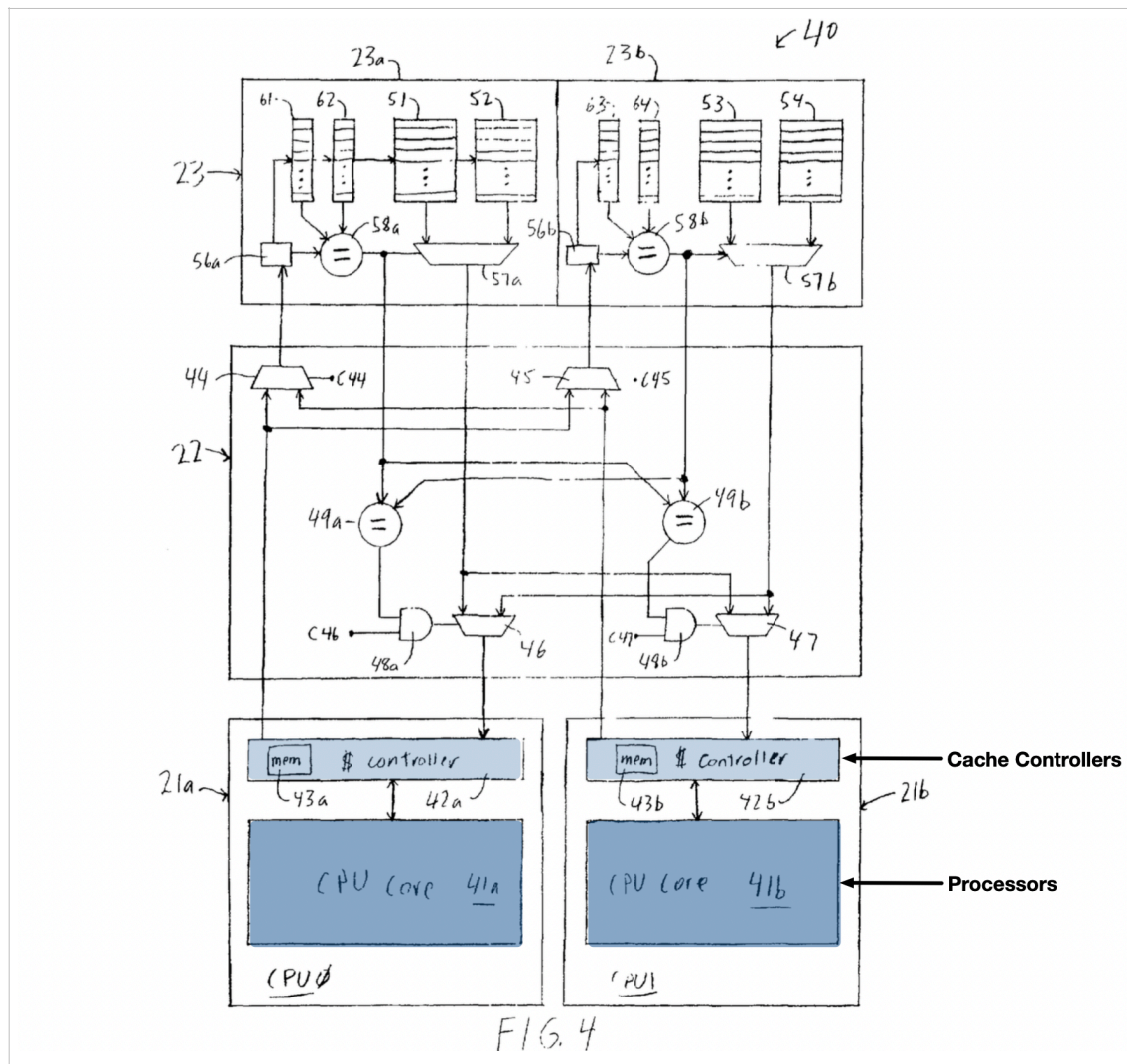
Some manufactures combine two or more central processing units (CPUs) on a single chip and sell the chip as a multi-processor unit (MPU). The MPU takes advantage of parallel processing to increase performance over a single CPU. An MPU typically includes a cache memory to store data in anticipation of future use by the CPUs. The cache memory is smaller and faster than the MPU's main memory, and thus can transfer data to the CPUs in much less time than data from the main memory. When data requested by the CPUs is in the cache memory, there is a cache hit, and CPU performance approaches the speed of the cache memory. Conversely, when there is a cache miss, the requested data must be retrieved from main memory, and thus CPU performance approaches the speed of main memory. Thus, increased performance may be achieved by maximizing the percentage of cache hits during operation. [Cherabuddi, 1:20–27, emphasis added]

FIG. 2 shows an MPU 20 that is one embodiment of MPU 11 of FIG. 1. ***MPU 20 is shown to include first and second processors such as central processing units (CPUs) 21a-21b,*** a cache access circuit 22, and a dynamically allocated cache memory 23. CPUs 21a-21b are well-known processing devices. Cache memory 23 is partitioned into first and second cache memory partitions 23a-23b, and is preferably a high speed cache memory device such as SRAM, although other cache devices may be used. For the purpose of discussion herein, each cache memory partition 23a-23b is configured as a 2-way associative cache memory. Of course, in actual embodiments, the cache memory partitions may be configured for other levels of associativity. [Cherabuddi, 3:26–38, emphasis added]

(88) In the figure, Cherabuddi illustrates two processors, CPU0 and CPU1, each of which is capable of executing an instruction stream, the pair combined on a single chip, to produce a single-chip multi-processor unit. The quoted passages confirm this understanding. Note that a person of ordinary skill in the art would have understood an “instruction thread” and an “instruction stream” to mean the same thing.

b) a first processor, coupled to fetch instructions and access data through a first cache controller; a second processor, coupled to fetch instructions and access data through a second cache controller;

(89) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the processors, labeled *CPU Core 41a* and *CPU Core 41b*, are highlighted in blue, and the cache controllers, labeled *\$ Controller 42a* and *\$ Controller 42b*, are highlighted in lighter blue. Note that each claimed processor can be either a CPU core 41 or the entire CPU 21.



(90) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. ***Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and***

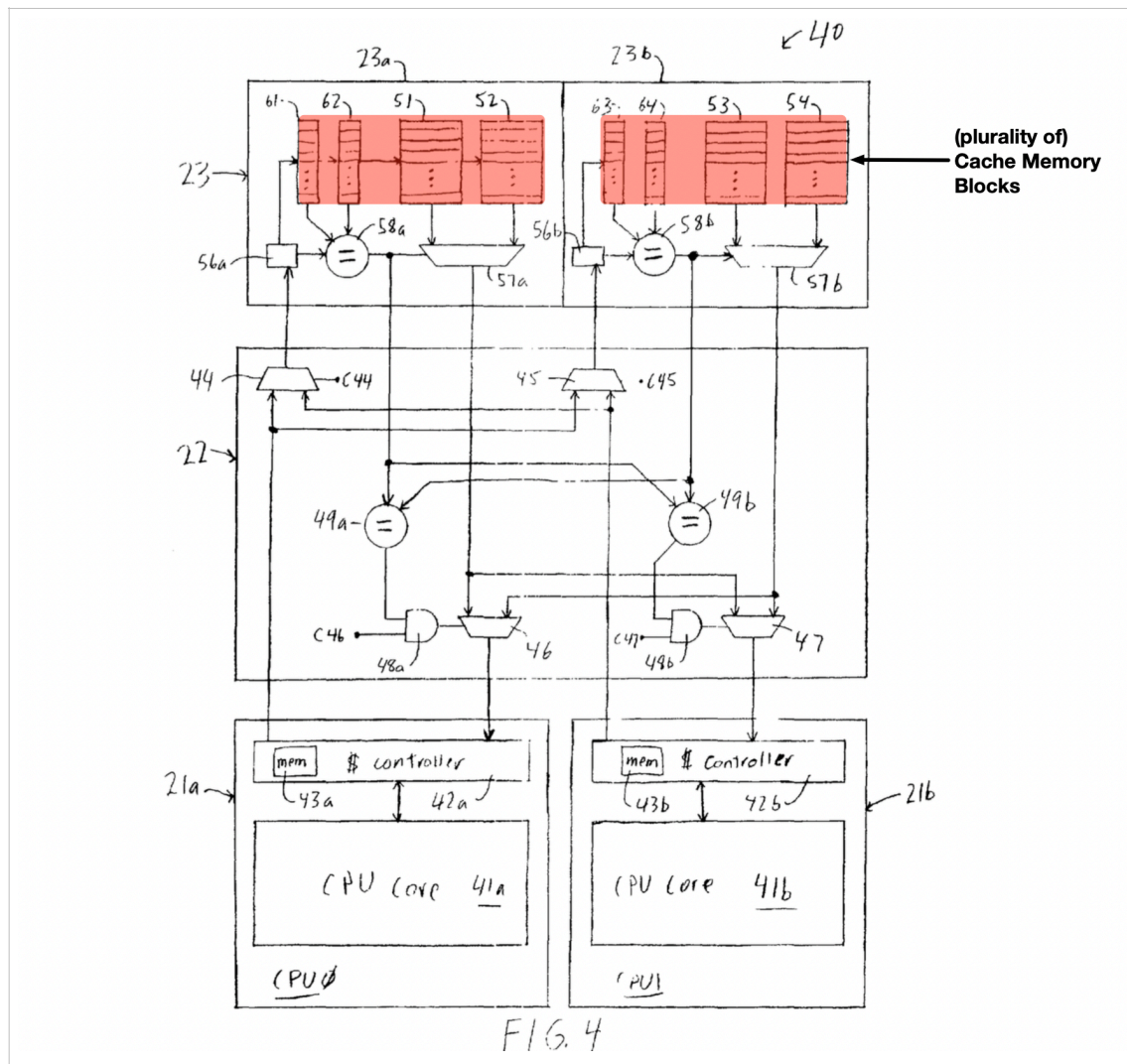
data between its associated CPU core 41 and cache access

circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

(91) Cherabuddi's description above confirms the understanding that, in the preceding figure, the processors, labeled *CPU Core 41a* and *CPU Core 41b*, are highlighted in blue, and the cache controllers, labeled *Controller 42a* and *Controller 42b*, are highlighted in lighter blue. The figure includes a first processor, dark blue, coupled to fetch instructions and access data through a first cache controller, light blue; a second processor, dark blue, coupled to fetch instructions and access data through a second cache controller, light blue. The passages above confirm the understanding that elements 41a and 41b are processors and that elements 42a and 42b are cache controllers. A person of ordinary skill would have understood that a processor accesses cached instructions and data through a cache controller (see the *Technical Background* section). In addition, the fact that cache 23 is a unified cache would have indicated to a person of ordinary skill that the cache holds both instructions and data (see the *Technical Background* section).

c) a plurality of cache memory blocks;

(92) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b. The limitation can be satisfied by the tag arrays coupled with the cache data blocks (indicated by highlighting the entire tag/data set in red for each half of the cache), and the limitation can also be satisfied by considering only the cache data blocks by themselves (elements 51, 52, 53, and 54). See, for example, '264 patent, 4:36–39, in which the patent describes that cache memory blocks 198 “incorporate[] data memory 200” and “tag memory 202.” Therefore, a person of skill would have understood that data arrays 51–52 and 53–54 coupled with tag arrays 61–62 and 63–64, respectively, constitute “a plurality of cache memory blocks,” as they include the entire tag/data set for each cache memory partition.



(93) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data

between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. Each data array 51-54 also includes a well-known address decoder (not shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at

a selected cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

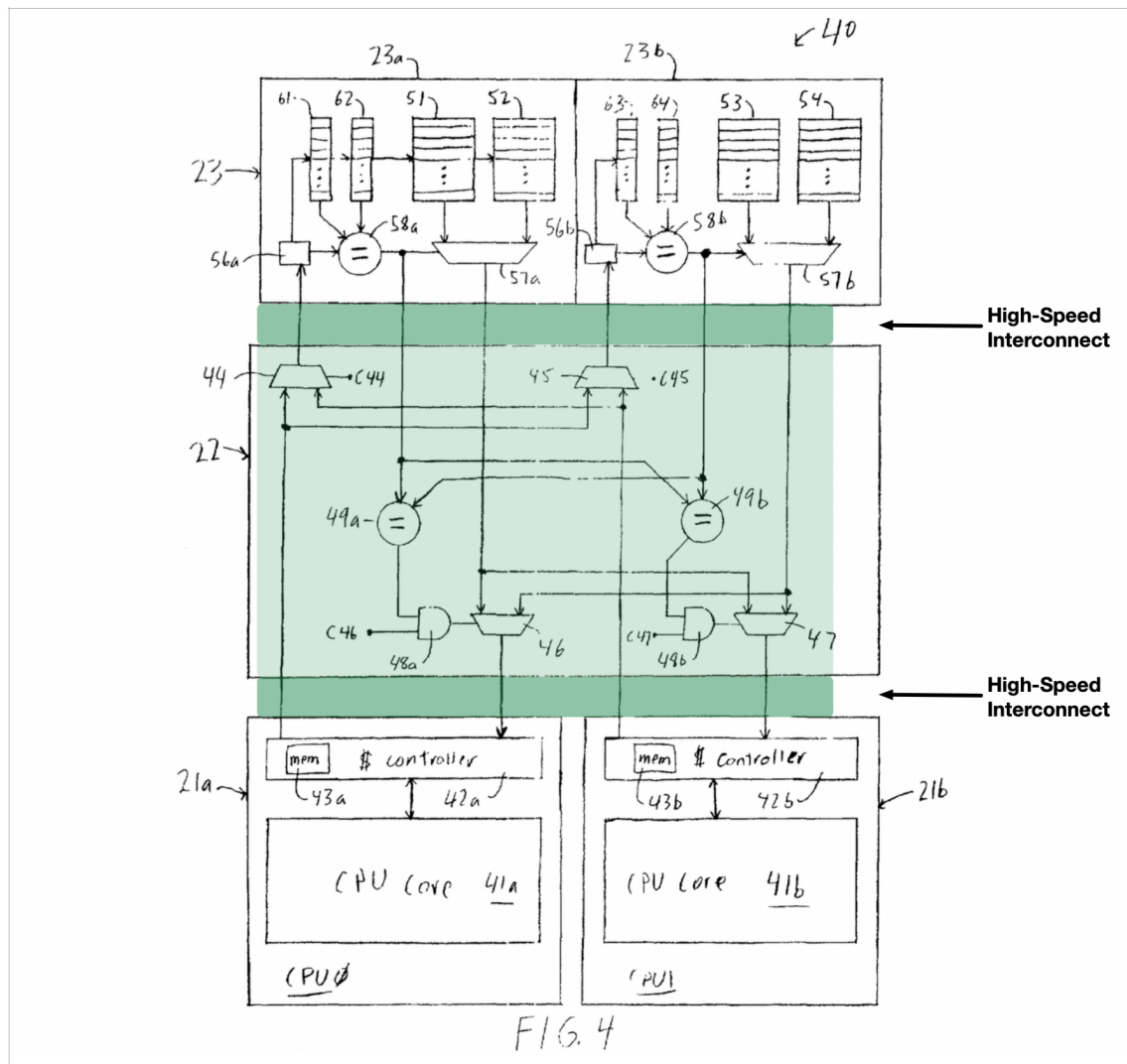
Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known. [Cherabuddi, 5:15–49, emphasis added]

(94) Cherabuddi's description above confirms the understanding that, in the preceding figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b. As mentioned, the limitation can be satisfied by coupling the tag arrays with the cache data blocks (indicated by the current highlighting in red, which includes the entire tag/data set, for each half of the cache), and the limitation can also be satisfied by considering only the cache data blocks by themselves (elements 51, 52, 53, and 54) as being "cache memory blocks." See previous comment regarding '264 patent, 4:36–39.

d) a high-speed interconnect coupling the plurality of cache memory blocks to the first and second cache controllers such that at least one allocable cache memory block is capable of being used by the first and second cache controllers; and

(95) As described in the *Technical Background* section, on a cache miss, data is fetched and transmitted back to the cache. This is done over wires.

(96) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the high-speed interconnect is highlighted in green. In the figure is shown the wires that directly connect the plurality of cache memory blocks to the first and second cache controllers. Further, the limitation “at least one allocable cache memory block is capable of being used” is satisfied by the cache being a cache ... e.g., it is not disabled or non-functional. Cherabuddi’s cache is designed to operate as a cache, e.g., it is designed for the cache blocks to be used.



(97) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers

address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on. [Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. Each data array 51-54 also includes a well-known address decoder (not

shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at a selected cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

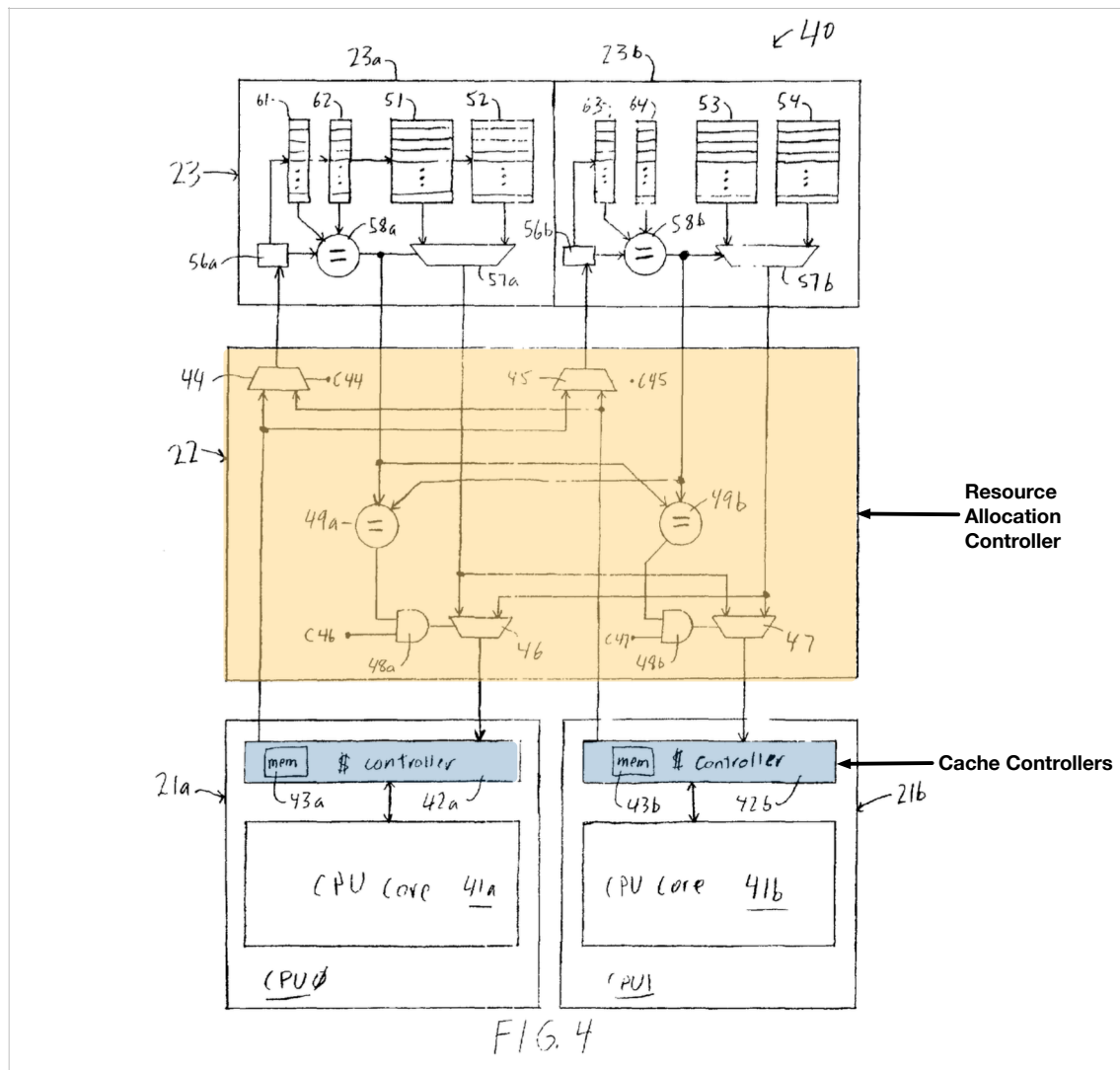
Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known.
[Cherabuddi, 5:15–49, emphasis added]

(98) Cherabuddi's description above confirms the understanding that, in the preceding figure, the high-speed interconnect is highlighted in green. In the figure is shown the wires that directly connect the plurality of cache memory blocks to the first and second cache controllers. The figure illustrates, and the text explains, that the high-speed interconnect couples the cache memory blocks to the cache controllers via MUXes such that both controllers have access to all of the cache memory blocks, depending on the address comparisons that drive the MUX select signals.

e) a resource allocation controller coupled to determine an accessing cache memory controller selected from the group consisting of the first and second cache memory controllers, whereby the accessing cache memory controller is allowed to access the allocable cache memory block,

(99) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the resource allocation controller is highlighted in gold. In the figure is shown Cherabuddi's cache access circuit 22, which determines which cache memory controller (the cache memory controllers are highlighted in blue) is allowed to access the cache blocks in partitions 23a and 23b. The claim describes explicitly what a "resource allocation controller" does (it "determine[s] an accessing cache memory controller selected from the group consisting of the first and second cache memory controllers, whereby the accessing cache memory controller is allowed to access the allocable cache memory block"). In addition, the Abstract of the '264 patent says that the resource allocation controller "determines which cache memory controller has access to the dynamically allocable cache memory block."

['264 patent, *Abstract*]



(100) Cherabuddi describes this as follows:

Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs

21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources. [Cherabuddi, 3:39–45, emphasis added]

Cache access circuit 22 is shown to include four multiplexers (MUXs) 44-47, two AND gates 48a and 48b, and two comparators 49a and 49b, although after reading this disclosure it will be evident to those skilled in the art that various other logic configurations may be used to selectively route addresses and data between MPU 20 and cache memory 23. **MUXes 44-45 selectively provide address information from CPUs 21a-21b to cache memory partitions 23a-23b, respectively, and MUXes 46-47 selectively provide data from cache memory partitions 23a-23b to CPUs 21a-21b, respectively.** MUXes 44-45 are controlled by control signals C44 and C45, respectively. MUX 46 is controlled by AND gate 48a, which includes a first input terminal coupled to receive a control signal C46 and a second input terminal coupled to comparator 49a. Comparator 49a includes input terminals coupled to receive select signals from comparators 58a and 58b of cache memory 23. MUX 47 is controlled by AND gate 48b, which includes a first input terminal coupled to receive a control signal C47 and a second input terminal coupled to comparator 49b. Comparator 49b includes input terminals coupled to receive select signals from comparators 58a and 58b of cache memory 23. Comparators 49a and 49b are well-known. Values for signals C44 and C46 may be stored in memory 43a of cache controller 42a, and values for signals C45 and C47 may be stored in memory 43b of cache controller 42b.

Specifically, MUX 44 selectively provides address and data information to cache memory partition 23a from either CPU

21a or CPU 21b in response to C44, and MUX 45 selectively provides address and data information to cache memory partition 23b from either CPU 21a or CPU 21b in response to C45. MUX 46 selectively returns data to CPU 21a from either cache memory partition 23a or 23b in response to AND gate 48a, and MUX 47 selectively returns data to CPU 21b from either cache memory partition 23a or 23b in response to AND gate 48b. [Cherabuddi, 5:50–6:18, emphasis added]

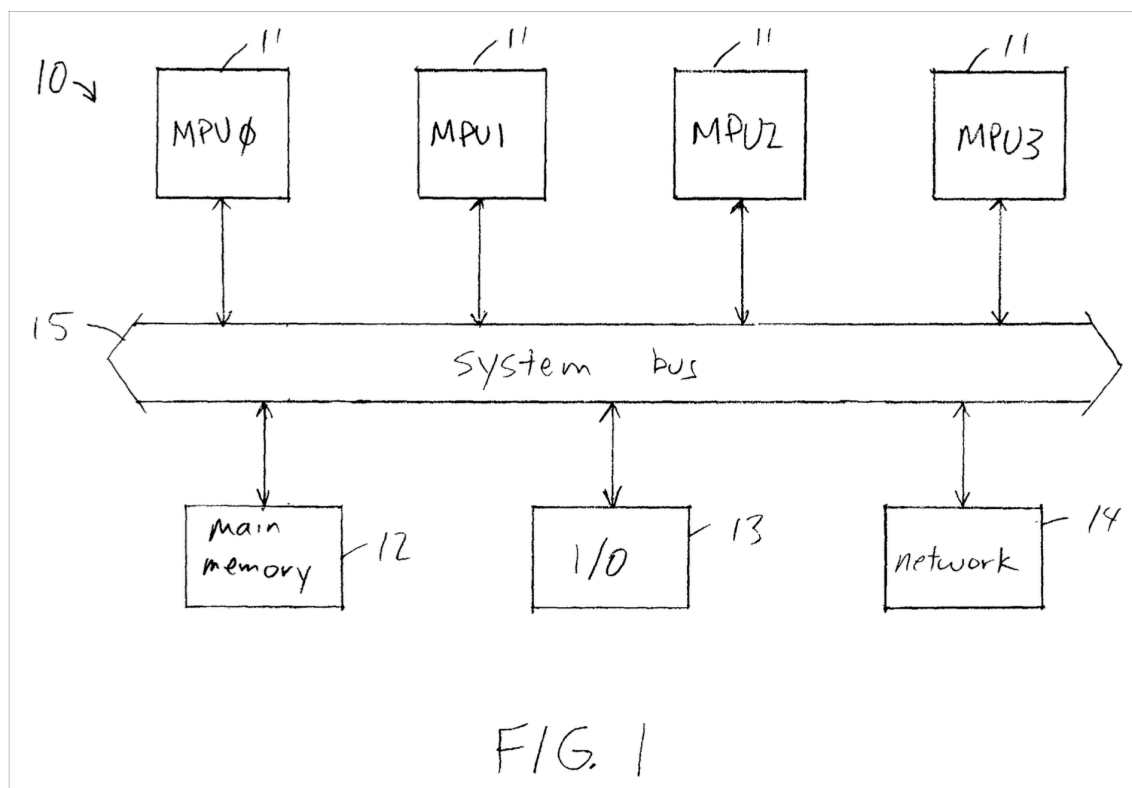
(101) Cherabuddi’s description above confirms the understanding that, in the preceding figure, the resource allocation controller is highlighted in gold. In the figure is shown Cherabuddi’s cache access circuit 22, which determines which cache memory controller (the cache memory controllers are highlighted in blue) is allowed to access the cache blocks in partitions 23a and 23b. For instance, Cherabuddi explains this dual-access arrangement, saying, “Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU’s processing requirements to more efficiently utilize cache resources.” [Cherabuddi, 3:39–45] The cache access circuit 22, highlighted in gold in the figure above, “selectively couples” the CPUs, by way of their cache controllers as shown in the figure, to the cache memory partitions 23a and 23b (corresponding to the claimed cache

memory blocks). The figure and description show that all allocable cache memory blocks are capable of being used by the first and second cache controllers, and therefore at least one allocable cache memory block is capable of being used by the first and second cache controllers, which means that, given a particular CPU (cache memory controller) and a targeted cache memory block that it is allowed to use, the accessing cache memory controller is allowed to access the allocable cache memory block.

f) wherein the cache memory blocks are usable by the cache controllers to store data and instructions fetched from a random-access memory.

(102) As described in the *Technical Background* section, caches store data and instructions fetched from another memory level, such as a main memory or another level of cache. This is what a cache's memory blocks are used for: they house the data and instructions. The operations are performed under the control of a cache controller that does the fetching and, among other functions, keeps the cache consistent and coherent. As illustrated in the *Technical Background* section, split caches hold either instructions or data, and unified caches hold both instructions and data, often just called 'data' because instructions *are* data — they are data words that are interpreted as instructions.

(103) In Figure 1, Cherabuddi discloses that the MPU shown in Figures 2 and 4 above is part of a system with main memory, and that the cache memory blocks store data and instructions fetched from that main memory, a random-access memory such as DRAM:



(104) Cherabuddi describes this as follows:

Some manufactures combine two or more central processing units (CPUs) on a single chip and sell the chip as a multi-processor unit (MPU). The MPU takes advantage of parallel processing to increase performance over a single CPU. *An MPU typically includes a*

cache memory to store data in anticipation of future use by the CPUs. The cache memory is smaller and faster than the MPU's main memory, and thus can transfer data to the CPUs in much less time than data from the main memory. When data requested by the CPUs is in the cache memory, there is a cache hit, and CPU performance approaches the speed of the cache memory. Conversely, **when there is a cache miss, the requested data must be retrieved from main memory**, and thus CPU performance approaches the speed of main memory. Thus, increased performance may be achieved by maximizing the percentage of cache hits during operation. [Cherabuddi, 1:20–27, emphasis added]

FIG. 1 shows a computer system 10 within which embodiments of the present invention may be implemented. System 10 is shown to include four MPUs 11 connected to each other and to a main memory 12, an input/output (I/O) device 13, and a network 14 via a system bus 15. Main memory 12 is shared by MPUs 11, and may be any suitable random access memory (RAM) such as, for example, DRAM. I/O device 13 allows a user to interact with system 10, and may include, for example, a computer monitor, keyboard, and/or mouse input. Network 14 may be any suitable network such as, for example, a local area network, a wide area network, and/or the Internet. Additional devices may be connected to the system bus 15 as desired. [Cherabuddi, 3:13–25, emphasis added]

(105) Cherabuddi's description above confirms the understanding that, in the preceding figure, the MPU shown in Figures 2 and 4 above is part of a system with

main memory, and that the cache memory blocks store data and instructions fetched from that main memory, a random-access memory such as DRAM.

(106) Further, a person of ordinary skill would have understood what is described in the *Technical Background* section, which is that caches store data and instructions fetched from another memory level, such as a main memory or another level of cache. This is what a cache's memory blocks are used for: they house the data and instructions. The operations are performed under the control of a cache controller that does the fetching and, among other functions, keeps the cache consistent and coherent. A person of ordinary skill would have known all of this. A person of ordinary skill would also have understood the concept illustrated in the *Technical Background* section, that split caches hold either instructions or data, and unified caches hold both instructions and data, often just called 'data' because instructions *are* data — they are data words that are interpreted as instructions. Cherabuddi illustrates a unified cache in Figure 4, which would have informed a person of ordinary skill that the cache would hold both instructions and data. Therefore, a person of ordinary skill would have understood that Cherabuddi's cache memory blocks are usable by the cache controllers to store data and instructions fetched from a random-access memory.

(107) It is my opinion that Cherabuddi discloses all the elements of Claim 1 and thus renders the claim obvious.

2. Claim 2

(108) Claim 2 reads as follows:

2. The processor integrated circuit of claim 1, further comprising a plurality of first level cache systems, wherein the first processor fetches instructions and accesses data from the first cache controller through a first first level cache system, and wherein the second processor fetches instructions and accesses data from the second cache controller through a second first level cache system.

(109) The following sections analyze this claim in light of the prior art.

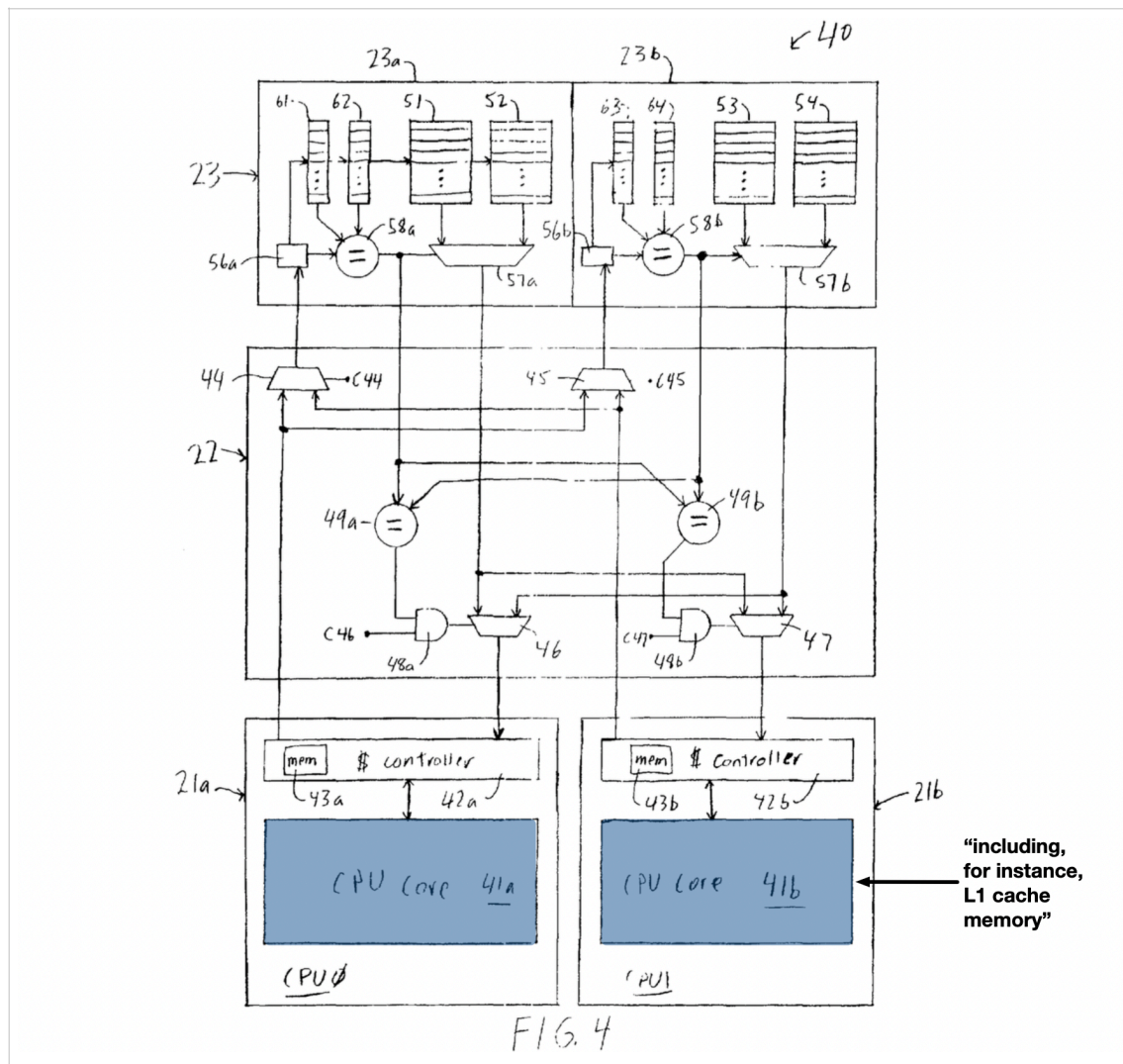
a) The processor integrated circuit of claim 1,

(110) As shown above, the limitations for Claim 1 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) further comprising a plurality of first level cache systems, wherein the first processor fetches instructions and accesses data from the first cache controller through a first first level cache system, and wherein the second processor fetches instructions and accesses data from the second cache controller through a second first level cache system.

(111) As described in the *Technical Background* section, processors access second-level caches after first looking in a first-level cache; thus, the second-level cache is accessed “through” the first-level cache system.

(112) Cherabuddi discloses that the CPU Cores 41a and 41b in Figure 4 contain first-level caches. In the figure below, the CPU Cores, which contain the first-level (Level-1, or L1) caches, are highlighted in blue.



(113) Cherabuddi describes this as follows:

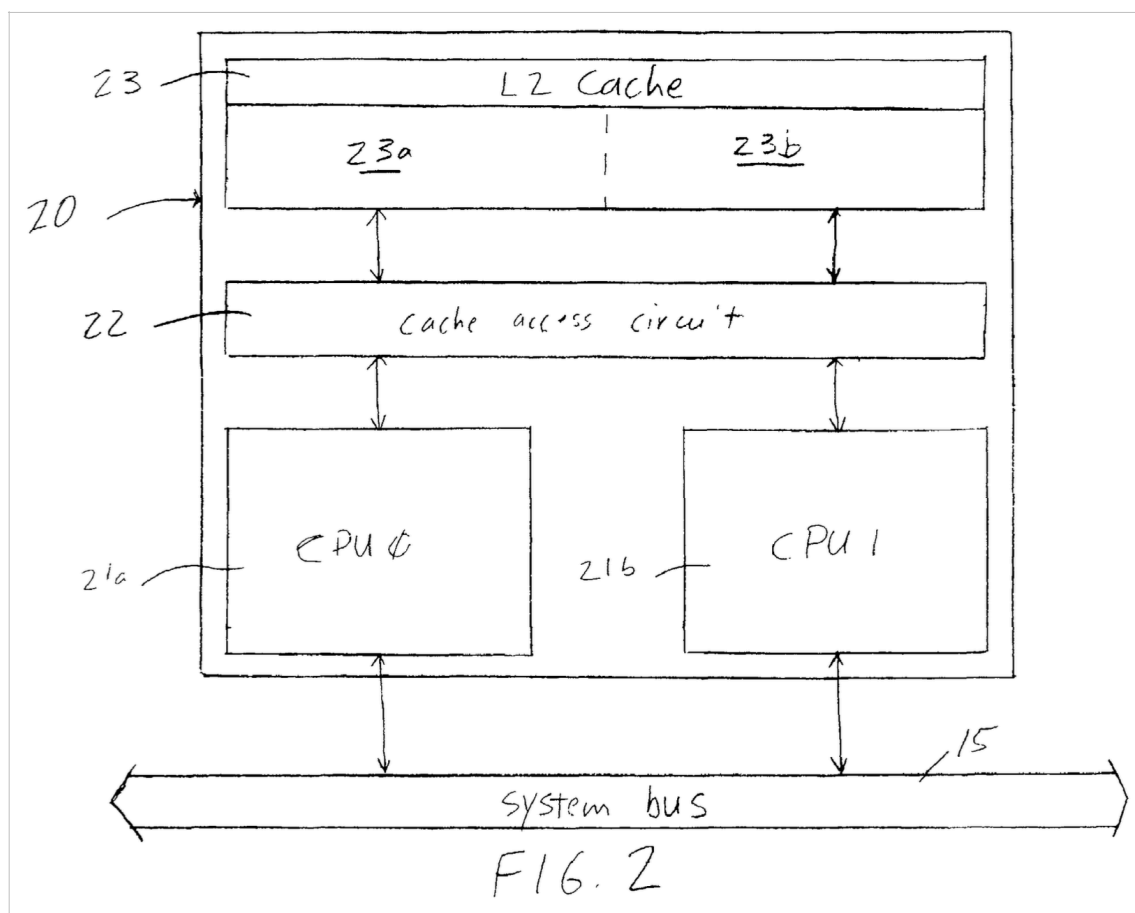
FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. ***CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory***, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

(114) Cherabuddi confirms that the CPU Cores contain L1 caches (first-level caches), because Cherabuddi specifically says so in the passage above: “CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory.” In addition, a person of ordinary skill would have known that L1 caches exist because the cache partitions 23a and 23b make up what Cherabuddi calls the *L2 Cache*, meaning that the CPUs each contain L1 caches.

(115) Further, a person of ordinary skill would have known, and as described in the *Technical Background* section, processors access second-level caches after first looking in a first-level cache; thus, a second-level cache is accessed “through” the

first-level cache system. Moreover, accessing an L2 cache is done by making requests of that cache's controller. The L2 cache has two controllers: one connected to CPU core 41a, and another connected to CPU core 41b. This means that, in Cherabuddi's system, the CPUs fetch instructions and access data from the first/second cache controllers through their first level cache systems.

(116) The L2 cache is shown in Figure 4 above and is also shown in Figure 2 below:



(117) This further indicates what a person of ordinary skill would have understood reading Cherabuddi: that the system discloses a plurality of first level cache systems, one in each of the processors (CPUs 21a and 21b), and that each processor fetches instructions and accesses data from the L2 cache (via the *controller* of the L2 cache), after first checking the first-level cache system within the CPU. Thus, a person of ordinary skill would have understood that each processor fetches instructions and accesses data from its associated cache controller through the processor's first level cache system.

(118) It is my opinion that Cherabuddi discloses all the elements of Claim 2 and thus renders the claim obvious.

3. Claim 3

(119) Claim 3 reads as follows:

3. The processor integrated circuit of claim 1, wherein the cache memory blocks further comprise cache tag memory.

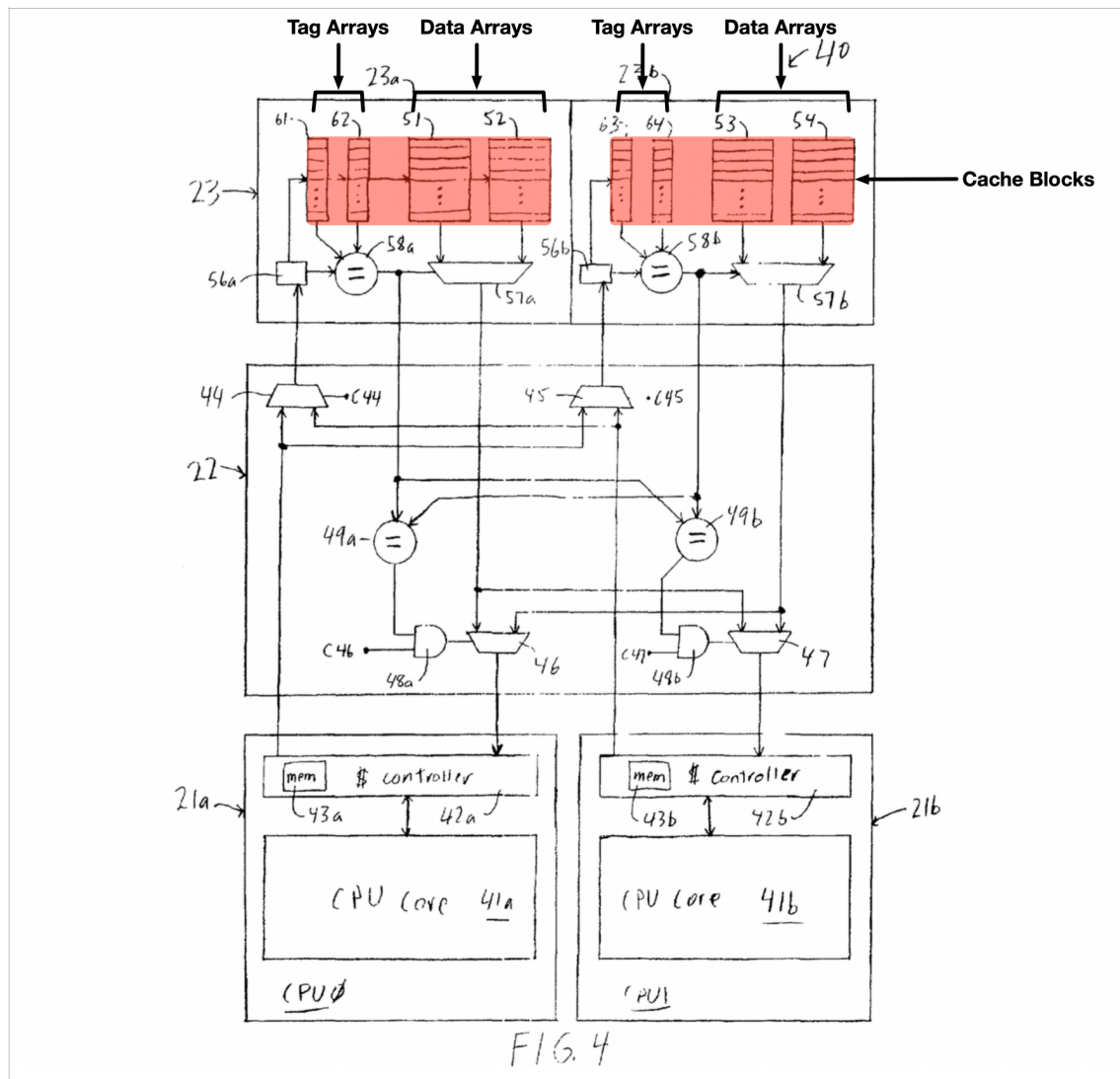
(120) The following sections analyze this claim in light of the prior art.

a) The processor integrated circuit of claim 1,

(121) As shown above, the limitations for Claim 1 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein the cache memory blocks further comprise cache tag memory.

(122) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks that comprise cache tag memory. The tag memory blocks are labeled as elements 61 and 62 in partition 23a (left side), and elements 63 and 64 in partition 23b (right side).



(123) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data

between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. Each data array 51-54 also includes a well-known address decoder (not shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at a selected

cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known. [Cherabuddi, 5:15–49, emphasis added]

(124) Cherabuddi's description above confirms the understanding that, in the preceding figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks that comprise cache tag memory. The tag memory blocks are labeled as elements 61 and 62 in partition 23a (left side), and elements 63 and 64 in partition 23b (right side). Cherabuddi explicitly describes the elements as tag-memory arrays in the passage above: "Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable ***tag arrays*** 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable ***tag arrays*** 63-64, respectively." Therefore, the cache memory blocks (elements highlighted in red in the figure above) comprise cache tag memory.

(125) It is my opinion that Cherabuddi discloses all the elements of Claim 3 and thus renders the claim obvious.

4. Claim 5

(126) Claim 5 reads as follows:

5. A method of dynamically allocating cache on a multiple-processor integrated circuit, where the multiple processor integrated circuit is used in a partitionable multiple-processor system and comprises:

a plurality of processors each coupled to receive instructions from a first level cache associated therewith,

a plurality of allocable upper level cache memory blocks, interconnect apparatus for transmitting cache misses at each first level cache to upper level cache memory blocks assigned thereto, and

allocation apparatus for assigning upper level cache memory blocks to processors;

the method comprising the steps of:

monitoring past cache performance associated with processors and partitions;

determining desired processor to partition and upper level cache block allocations to processors; and

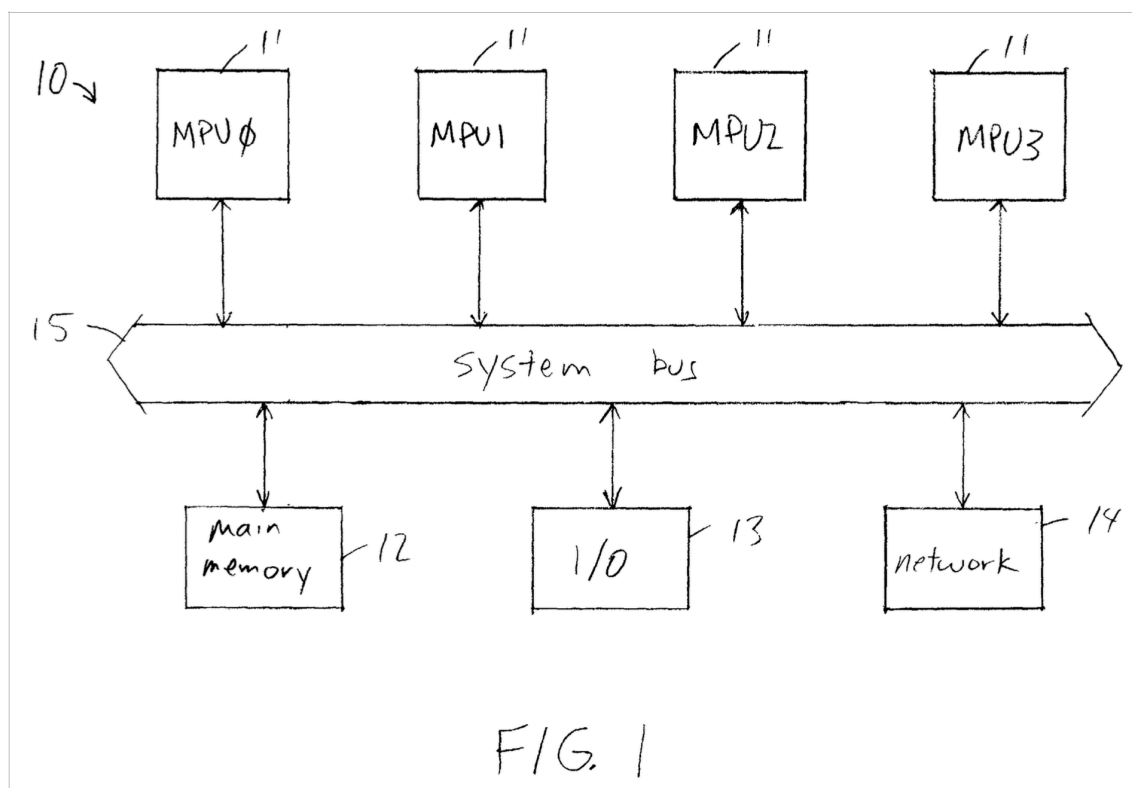
repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits,

wherein the upper level cache blocks are usable to store data and instructions fetched from a random-access main memory.

(127) The following sections analyze this claim in light of the prior art.

a) A method of dynamically allocating cache on a multiple-processor integrated circuit, where the multiple processor integrated circuit is used in a partitionable multiple-processor system and comprises:

(128) Cherabuddi discloses a dynamically allocatable/partitionable cache on a multiple-processor integrated circuit in a multiple-processor system. The multiple-processor system is shown below:



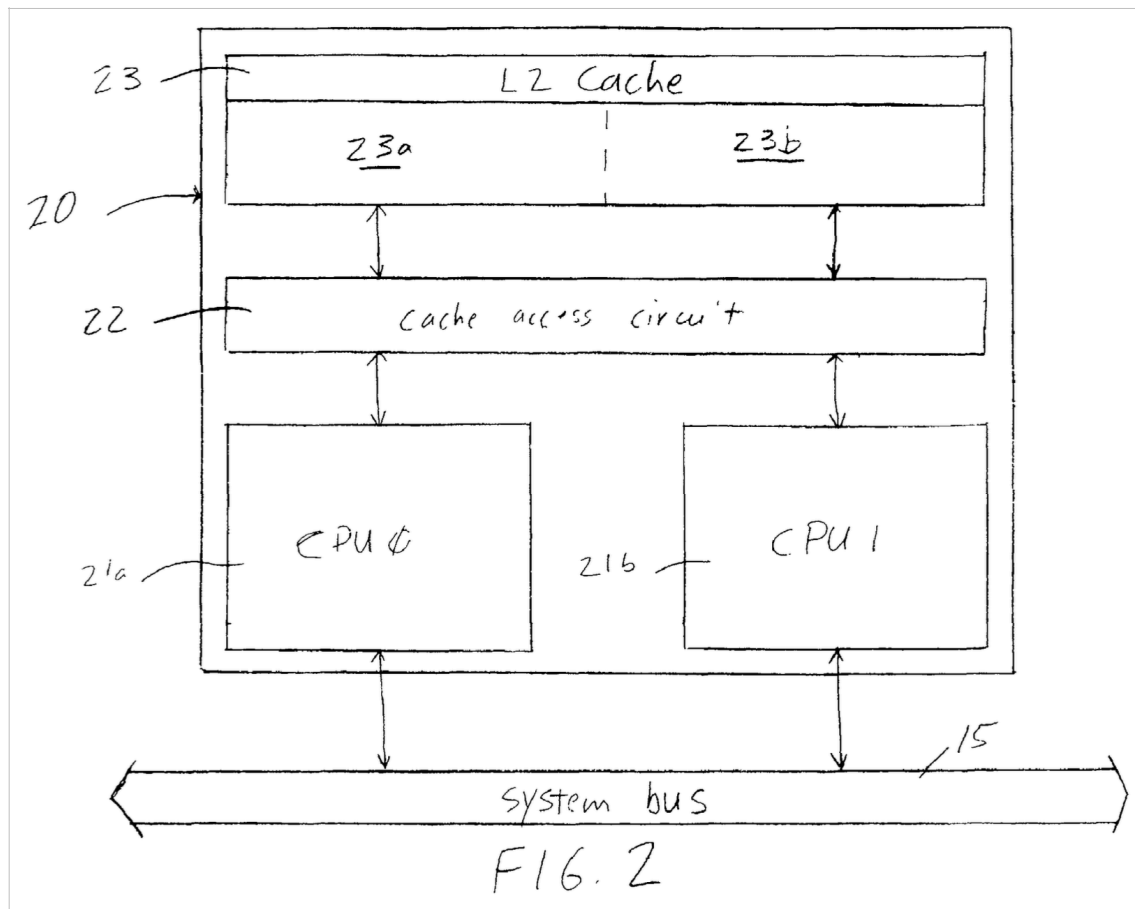
(129) Cherabuddi describes the system as follows:

FIG. 1 shows a computer system 10 within which embodiments of the present invention may be implemented.

System 10 is shown to include four MPUs 11 connected to each other and to a main memory 12, an input/output (I/O) device 13, and a network 14 via a system bus 15. Main memory 12 is shared by MPUs 11, and may be any suitable random access memory (RAM) such as, for example, DRAM. I/O device 13 allows a user to interact with system 10, and may include, for example, a computer monitor, keyboard, and/or mouse input. Network 14 may be any suitable network such as, for example, a local area network, a wide area network, and/or the Internet. Additional devices may be connected to the system bus 15 as desired. [Cherabuddi, 3:13–25, emphasis added]

(130) Cherabuddi’s description above (e.g., “System 10 is shown to include four MPUs 11 connected to each other and to a main memory 12”) confirms the understanding that the preceding figure shows a multiple-processor system.

(131) The dynamically allocatable/partitionable cache on a multiple-processor integrated circuit is shown below:



(132) Cherabuddi describes this as follows:

FIG. 2 shows an MPU 20 that is one embodiment of MPU 11 of FIG. 1. MPU 20 is shown to include first and second processors such as central processing units (CPUs) 21a-21b, a cache access circuit 22, and a dynamically allocated cache memory 23.

CPUs 21a-21b are well-known processing devices. **Cache memory 23 is partitioned into first and second cache memory partitions 23a-23b**, and is preferably a high speed cache memory device such as SRAM, although other cache devices may be used. For the purpose of

discussion herein, each cache memory partition 23a-23b is configured as a 2-way associative cache memory. Of course, in actual embodiments, the cache memory partitions may be configured for other levels of associativity.

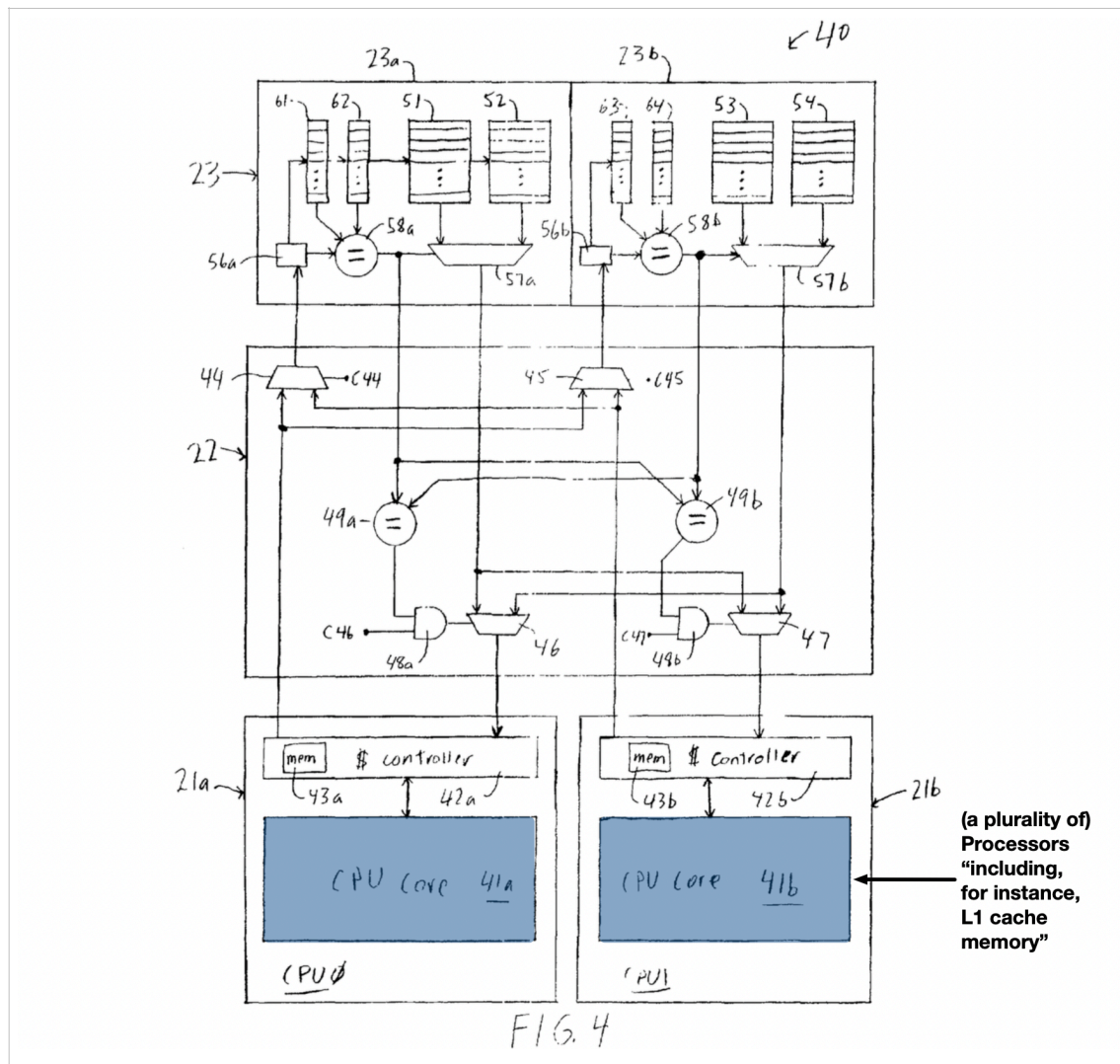
Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources. [Cherabuddi, 3:26–45, emphasis added]

(133) Cherabuddi's description above (e.g., "Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources") confirms the understanding that the preceding figure shows a dynamically allocatable/partitionable cache on a multiple-processor integrated circuit.

b) a plurality of processors each coupled to receive instructions from a first level cache associated therewith,

(134) As illustrated in the *Technical Background* section, processors receive instructions from their first-level cache in order to operate.

(135) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the processors, labeled *CPU Core 41a* and *CPU Core 41b*, are highlighted in blue, and Cherabuddi indicates that each core contains first level (Level 1 or L1) caches. Note that each claimed processor can be either a CPU core 41 or the entire CPU 21.



(136) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. **Each CPU 21 is shown to include a CPU core 41** and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its

associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. ***CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory***, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

(137) In the figure and passage above, Cherabuddi discloses a plurality of processors, CPU cores 41a and 41b, highlighted in blue.

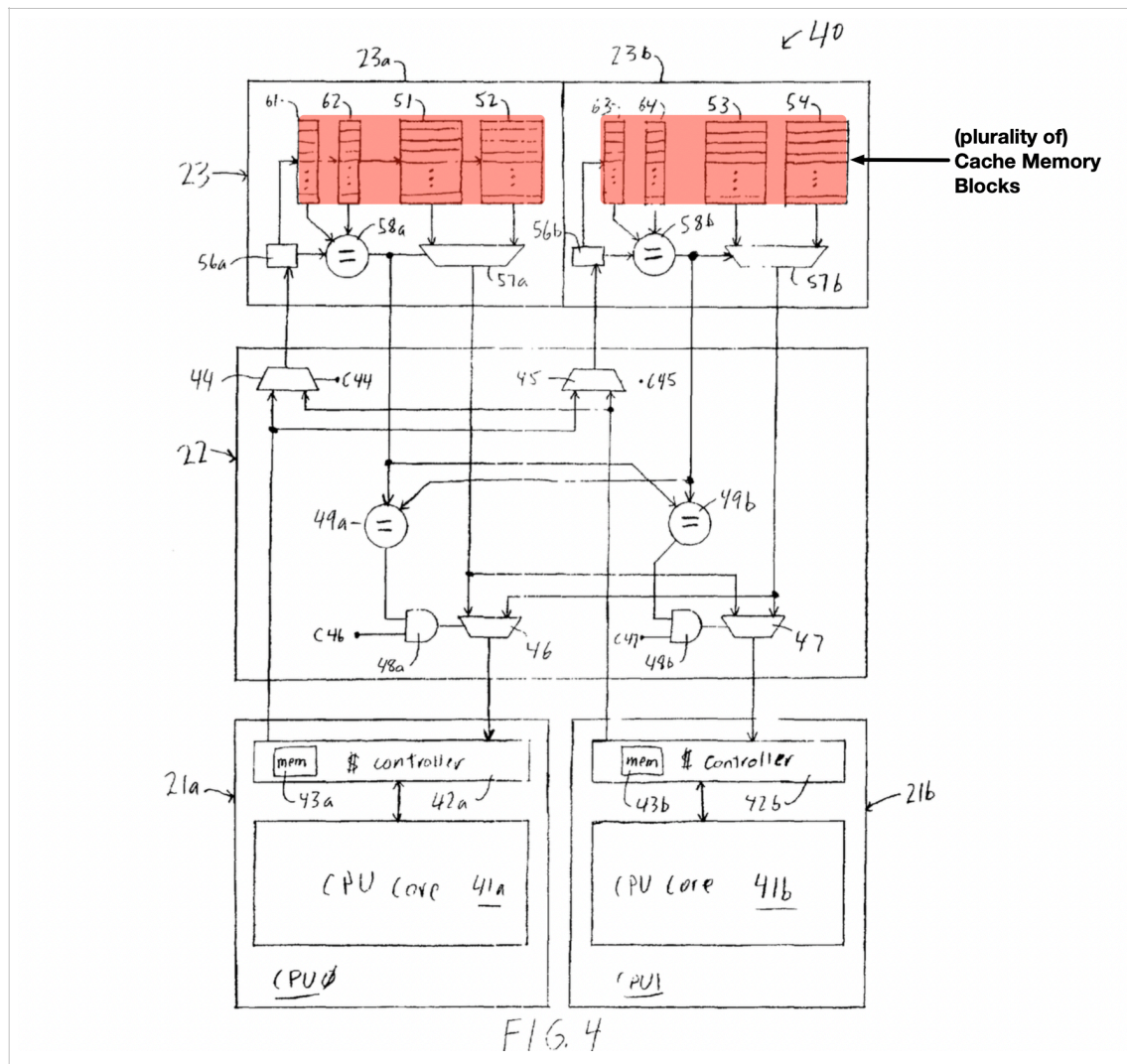
(138) In the passage above, Cherabuddi explicitly discloses that each CPU core (41a and 41b) contains L1 cache, e.g., “CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory.”

(139) Cherabuddi’s description above confirms the understanding that, in the preceding figure, the processors, labeled *CPU Core 41a* and *CPU Core 41b*, are highlighted in blue, and Cherabuddi indicates that each core contains first level (Level 1 or L1) caches. Further, a person of ordinary skill would have known that, as illustrated in the *Technical Background* section, processors receive instructions from their first-level cache in order to operate.

c) a plurality of allocable upper level cache memory blocks,

(140) In this context, and as is consistent with the '264 patent, “allocable” means “available” (e.g., can be reached and used by the processors), and “upper level” means “further away from the processor” (e.g., L2 cache).

(141) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b. The limitation can be satisfied by the tag arrays coupled with the cache data blocks (indicated by highlighting the entire tag/data set in red for each half of the cache), and the limitation can also be satisfied by considering only the cache data blocks by themselves (elements 51, 52, 53, and 54). See, for example, '264 patent, 4:36–39.



(142) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data

between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. Each data array 51-54 also includes a well-known address decoder (not shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at

a selected cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

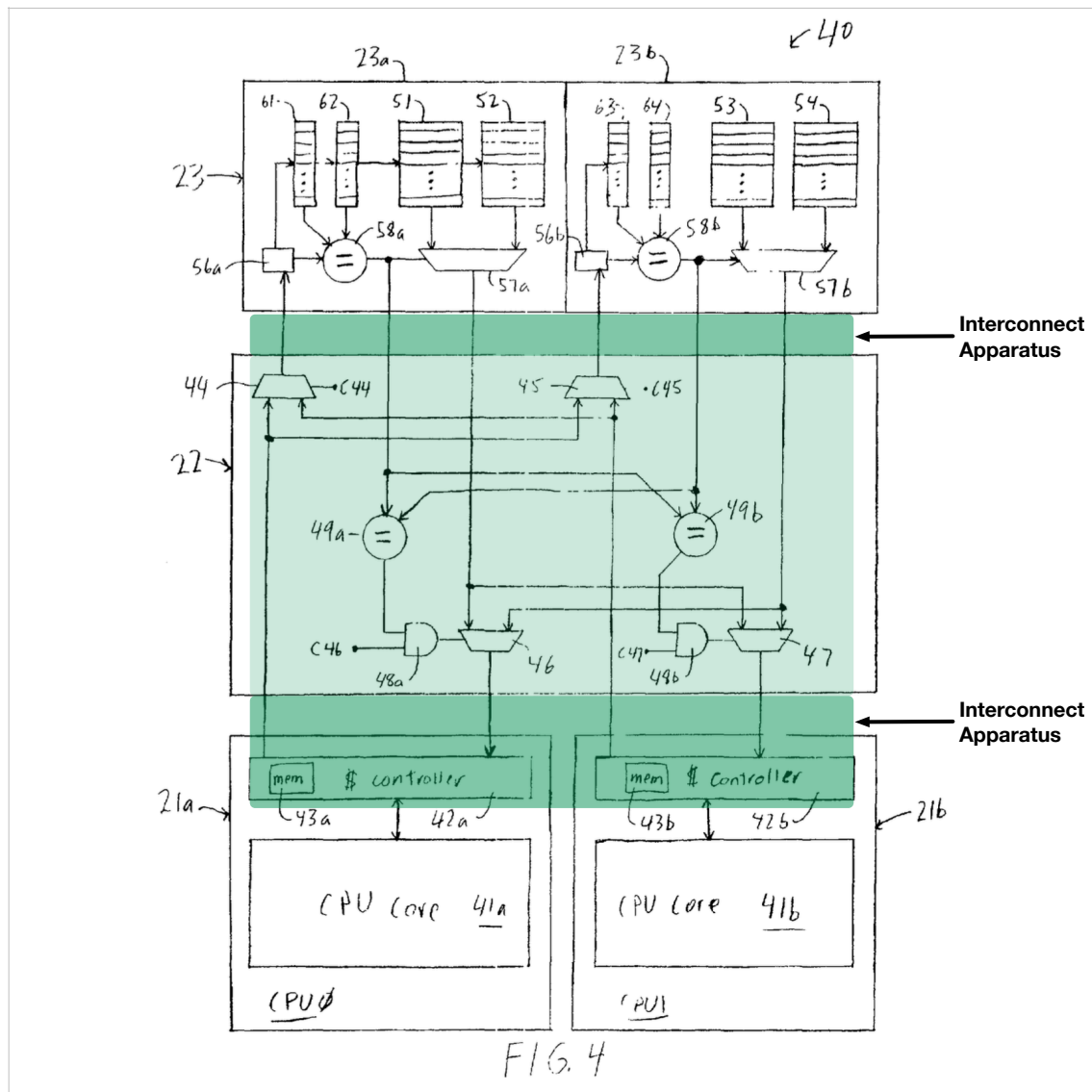
Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known. [Cherabuddi, 5:15–49, emphasis added]

(143) Cherabuddi's description above confirms the understanding that, in the preceding figure, the plurality of cache memory banks is highlighted in red. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b. As mentioned, the limitation can be satisfied by coupling the tag arrays with the cache data blocks (indicated by the current highlighting in red, which includes the entire tag/data set, for each half of the cache), and the limitation can also be satisfied by considering only the cache data blocks by themselves (elements 51, 52, 53, and 54) as being "cache memory blocks."

d) interconnect apparatus for transmitting cache misses at each first level cache to upper level cache memory blocks assigned thereto, and

(144) As described in the *Technical Background* section, on a cache miss, data must be fetched and transmitted back to the cache. This is done over wires.

(145) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the interconnect apparatus is highlighted in green. In the figure is shown the wires that directly connect the plurality of cache memory blocks to the first and second cache controllers, as well as the L2 cache controllers themselves.



(146) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller

42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on. [Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. **Each data**

array 51-54 also includes a well-known address decoder (not shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at a selected cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

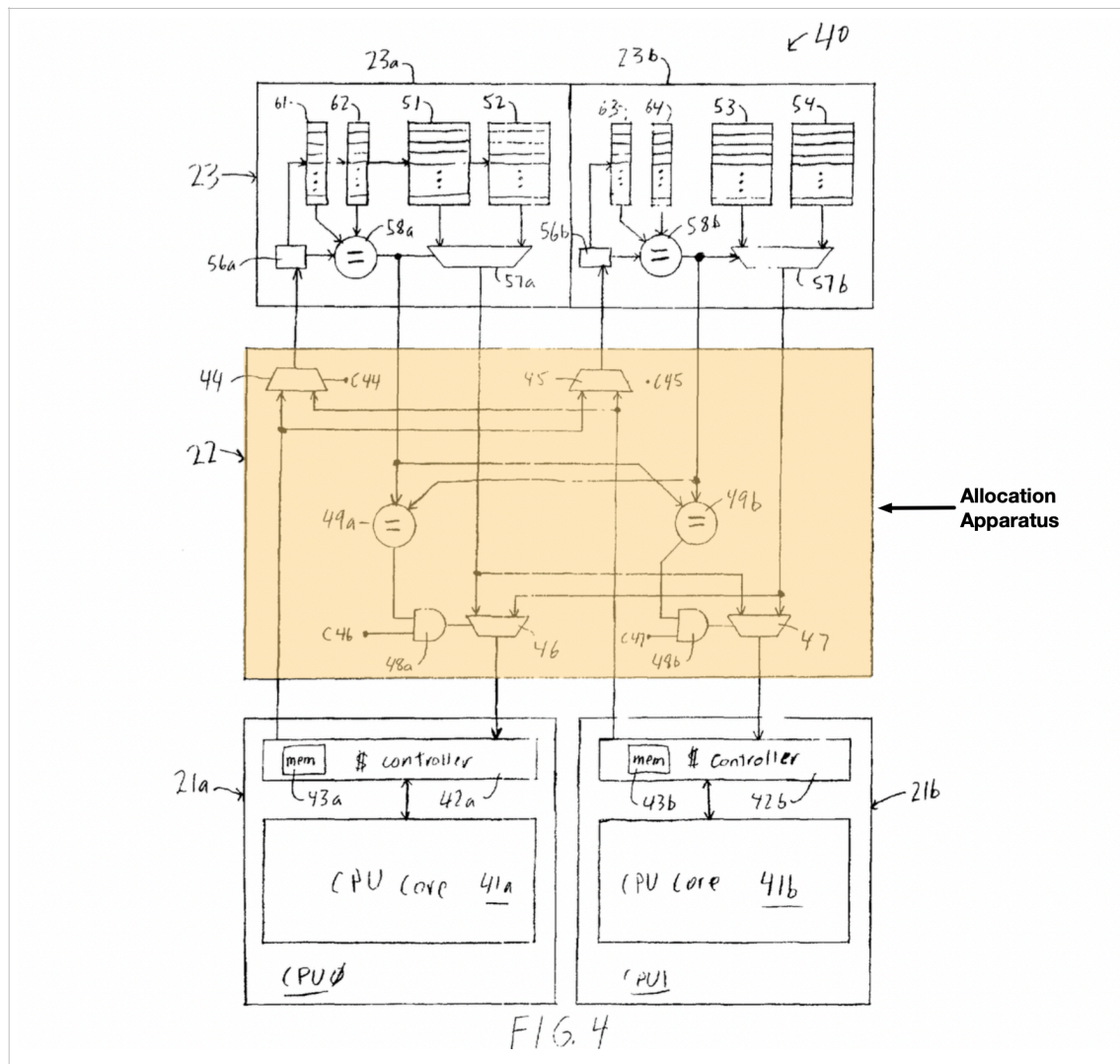
Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known.

[Cherabuddi, 5:15–49, emphasis added]

(147) Cherabuddi's description above confirms the understanding that, in the preceding figure, the high-speed interconnect is highlighted in green. In the figure is shown the wires that directly connect the plurality of cache memory blocks to the first and second cache controllers, as well as the L2 cache controllers themselves. The figure illustrates, and the text explains, that the high-speed interconnect couples the cache memory blocks to the cache controllers via MUXes such that both controllers have access to all of the cache memory blocks, depending on the address comparisons that drive the MUX select signals.

e) allocation apparatus for assigning upper level cache memory blocks to processors;

(148) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the resource allocation controller is highlighted in gold. In the figure is shown Cherabuddi's cache access circuit 22, which determines which cache memory controller is allowed to access the cache blocks in partitions 23a and 23b. In other words, it determines which upper level cache memory blocks have been assigned to which processors and is thus a claimed "allocation apparatus."



(149) Cherabuddi describes this as follows:

Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs

21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources. [Cherabuddi, 3:39–45, emphasis added]

Cache access circuit 22 is shown to include four multiplexers (MUXs) 44-47, two AND gates 48a and 48b, and two comparators 49a and 49b, although after reading this disclosure it will be evident to those skilled in the art that various other logic configurations may be used to selectively route addresses and data between MPU 20 and cache memory 23. **MUXes 44-45 selectively provide address information from CPUs 21a-21b to cache memory partitions 23a-23b, respectively, and MUXes 46-47 selectively provide data from cache memory partitions 23a-23b to CPUs 21a-21b, respectively.** MUXes 44-45 are controlled by control signals C44 and C45, respectively. MUX 46 is controlled by AND gate 48a, which includes a first input terminal coupled to receive a control signal C46 and a second input terminal coupled to comparator 49a. Comparator 49a includes input terminals coupled to receive select signals from comparators 58a and 58b of cache memory 23. MUX 47 is controlled by AND gate 48b, which includes a first input terminal coupled to receive a control signal C47 and a second input terminal coupled to comparator 49b. Comparator 49b includes input terminals coupled to receive select signals from comparators 58a and 58b of cache memory 23. Comparators 49a and 49b are well-known. Values for signals C44 and C46 may be stored in memory 43a of cache controller 42a, and values for signals C45 and C47 may be stored in memory 43b of cache controller 42b.

Specifically, MUX 44 selectively provides address and data information to cache memory partition 23a from either CPU

21a or CPU 21b in response to C44, and MUX 45 selectively provides address and data information to cache memory partition 23b from either CPU 21a or CPU 21b in response to C45. MUX 46 selectively returns data to CPU 21a from either cache memory partition 23a or 23b in response to AND gate 48a, and MUX 47 selectively returns data to CPU 21b from either cache memory partition 23a or 23b in response to AND gate 48b. [Cherabuddi, 5:50–6:18, emphasis added]

(150) Cherabuddi's description above confirms the understanding that, in the preceding figure, the allocation apparatus is highlighted in gold. In the figure is shown Cherabuddi's cache access circuit 22, which determines which cache memory controller (the cache memory controllers are highlighted in blue) is allowed to access the cache blocks in partitions 23a and 23b. For instance, Cherabuddi explains this dual-access arrangement, saying, "Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources." [Cherabuddi, 3:39–45] The cache access circuit 22, highlighted in gold in the figure above, "selectively couples" the CPUs, by way of their cache controllers as shown in the figure, to the cache memory partitions 23a and 23b (corresponding to the claimed cache

memory blocks). The figure and description confirm the understanding that the cache access circuit 22 is tasked with the job of assigning upper level cache memory blocks to processors.

f) the method comprising the steps of: monitoring past cache performance associated with processors and partitions; determining desired processor to partition and upper level cache block allocations to processors; and repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits,

(151) Cherabuddi describes the method of dynamically allocating cache on a multiple-processor integrated circuit in the passages below.

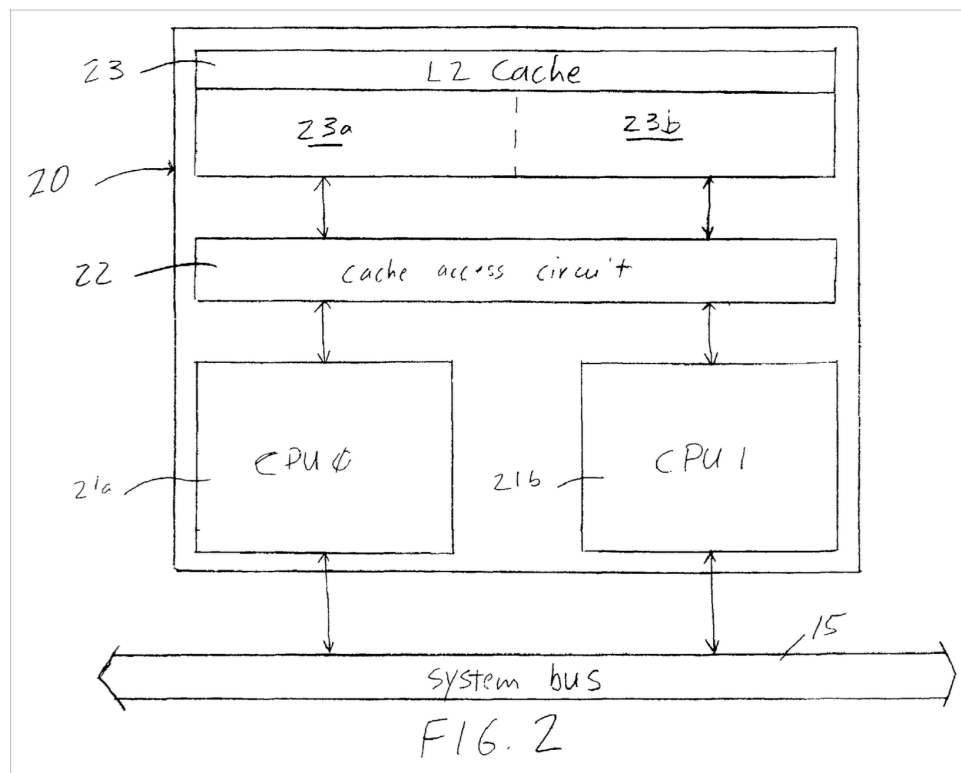
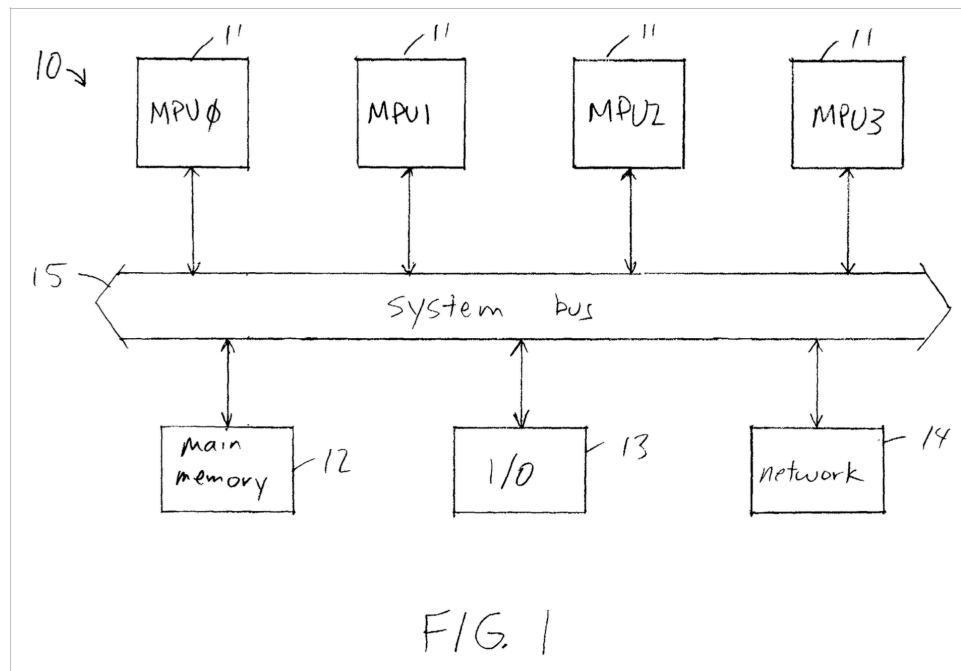


FIG. 2 shows an MPU 20 that is one embodiment of MPU 11 of FIG. 1. **MPU 20 is shown to include first and second processors such as central processing units (CPUs) 21a-21b, a cache access circuit 22, and a dynamically allocated cache memory 23.**

CPUs 21a-21b are well-known processing devices. **Cache memory 23 is partitioned into first and second cache memory partitions 23a-23b, and is preferably a high speed cache memory device such as SRAM**, although other cache devices may be used. For the purpose of discussion herein, each cache memory partition 23a-23b is configured as a 2-way associative cache memory. Of course, in actual embodiments, the cache memory partitions may be configured for other levels of associativity.

Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources.



Referring also to FIG. 1, *system 10 includes well-known system operating software that assigns tasks of one or more computer programs running thereon to the various MPUs 20 for execution. The operating software, which is often referred to as the system kernel, also assigns tasks between the CPUs 21a-21b of each MPU 20. For applications that include a single instruction execution thread and are thus best executed using only one CPU 21, e.g., for applications having a highly sequential instruction code, the kernel assigns all the tasks to one CPU and idles the other CPU. Conversely, for applications that can be divided into two parallel instruction execution threads, e.g., for applications having parallel execution loops, the kernel may assign different threads to CPUs 21a-21b for simultaneous execution therein.*

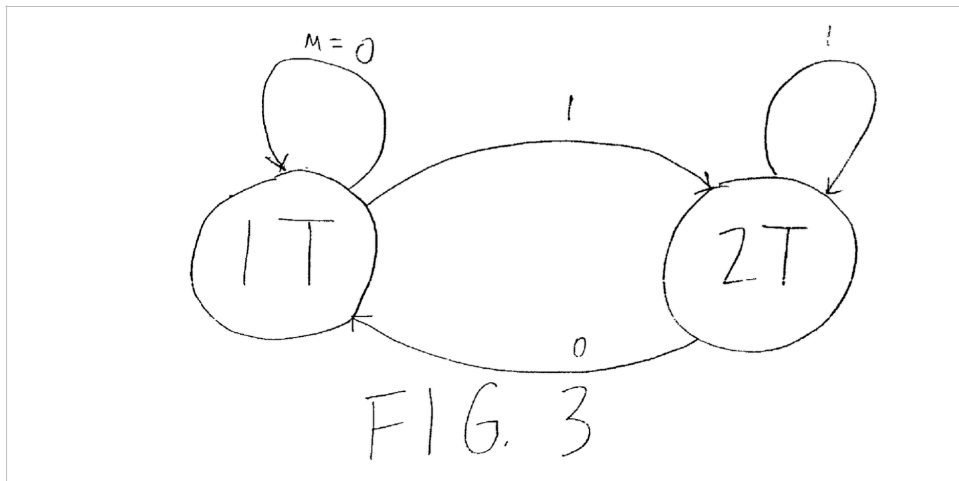


FIG. 3 illustrates state transitions of MPU 20 between a one-thread (1T) state and a two-thread (2T) state. In one embodiment, upon power-up of MPU 20, the kernel sets a mode signal $M=0$ to initialize MPU 20 to the 1T state. The kernel sets one of the CPUs 21 to an active state and sets the other CPU 21 to an inactive state. For purposes of discussion herein, *during the 1T state, the kernel sets CPU 21a as the active CPU and sets CPU 21b as the inactive CPU*, although in other embodiments the kernel may set CPU 21b as the active CPU and set CPU 21a as the inactive CPU. *While in the 1T state, the kernel assigns tasks of the computer program(s) only to the active CPU 21a, while the other CPU 21b remains idle. In response to $M=0$, cache access circuit 22 couples the first CPU 21a to both the first and second cache memory partitions 23a-23b to allow the first CPU 21a to use all resources of cache memory 23. In this state, the active CPU 21a may use cache memory partitions 23a-23b as a 4-way associative cache memory.*

If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition

MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21a-21b to the active state, and thereafter assigns one execution thread to CPU 21a and another execution thread to CPU 21b in a well-known manner. In response to M=1, dirty data in cache memory partition 23b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23b. ***The cache access circuit 22 couples the first CPU 21a to the first cache memory partition 23a for exclusive access thereto, and couples the second CPU 21b to the second cache memory partition 23b for exclusive access thereto. In this state, CPU 21a may use cache memory partition 23a as a dedicated 2-way associative cache memory, and CPU 21b may use cache memory partition 23b as a dedicated 2-way associative cache memory.***

Thereafter, if the kernel determines that only one of CPUs 21a-21b is necessary for a particular instruction code sequence, the kernel may transition MPU 20 to the 1T state by changing the mode signal to M=0, flushing the second cache memory partition 23b, and then assigning execution of the instruction code sequence to the active CPU 21a. [Cherabuddi, 3:26–4:37, emphasis added]

When MPU 20 is in the 2T state (e.g., when M=1), each CPU 21a-21b is processing its own instruction thread, and the kernel sets signals C44-C47 to logic low (i.e., logic 0) to simultaneously provide CPU 21a with exclusive use of cache memory partition 23a and to provide CPU 21b with exclusive use of cache memory partition 23b. Thus, C44=0 forces MUX 44 to provide an address or data from CPU 21a to cache memory partition 23a, C45=0

forces MUX 45 to provide an address or data from CPU 21b to cache memory partition 23b, C46=0 forces the output of AND gate 48a to logic 0 to force MUX 46 to provide data from cache memory partition 23a to CPU 21a, and C47=0 forces the output of AND gate 48b to logic 0 to force MUX 47 to provide data from cache memory partition 23b to CPU 21b.

To request data from cache memory partition 23a, CPU 21a provides a main memory address to address converter 56a via MUX 44. Address converter 56a converts the main memory address to a cache address that includes a tag address and a cache index. The cache index is used to select a cache line in data arrays 51-52 and associated tag arrays 61-62. If there is data stored at the selected cache line in data arrays 51 and/or 52, the data is read out to MUX 57a. Also, the tag fields from the selected line of tag arrays 61-62 are read out to comparator 58a, which also receives the tag address from address converter 56a. Comparator 58a compares the tag address with tag fields provided by tag arrays 61-62, and in response thereto provides a select signal to MUX 57 a that selects whether data from data array 51 or 52 (or neither, if there is no matching data) is read out to MUX 46 of cache access circuit 22. Since C46=0, MUX 46 provides matching data from cache memory partition 23a to cache controller 42a of CPU 21a.

CPU 21b may simultaneously request data from cache memory partition 23b in a similar manner. Thus, a main memory address provided by CPU 21b to address converter 56b via MUX 44 is converted into a cache address that includes a tag address and a cache index. The cache index selects a cache line in data arrays 53-54 and associated tag arrays 63-64.

If there is data stored at the selected cache line in data arrays 53 and/or 54, the data is read out to MUX 57b. Also, the tag fields from the selected line of tag arrays 63-64 are read out to comparator 58b, which also receives the tag address from address converter 56b. Comparator 58b compares the tag address with tag fields provided by tag arrays 63-64, and in response thereto provides a select signal to MUX 57b that selects whether data from data array 53 or 54 (or neither, if there is no matching data) is read out to MUX 47 of cache access circuit 22. *Since C47=0, MUX 47 provides matching data from cache memory partition 23b to cache controller 42b of CPU 21b.*

In this manner, CPU 21a may use cache memory partition 23a as a dedicated 2-way associative cache while CPU 21b simultaneously and independently uses cache memory partition 23b as a dedicated 2-way associative cache.

When MPU 20 transitions to the 1T state (e.g., M=0), the kernel sets CPU 21a as the active CPU and sets CPU 21b as the inactive CPU (as mentioned earlier, in other embodiments the kernel may set CPU 21b as the active CPU and set CPU 21a as the inactive CPU). The kernel also sets signal C44 to logic low and sets signals C45-C46 to logic high (i.e., logic 1) to provide CPU 21a with use of both cache memory partitions 23a-23b. Thus, C44=0 forces MUX 44 to provide an address or data from CPU 21a to cache memory partition 23a, C45=1 forces MUX 45 to provide the same address or data from CPU 21a to cache memory partition 23b, and C46=1 allows a result signal from comparator 49a to select whether data from cache memory partition 23a or 23b is returned to CPU 21a. Since CPU 21b is inactive, C47 is a don't care (d/c) for M=0.

To request data from both cache memory partitions 23a-23b, CPU 21a provides a main memory address to address converter 56a via MUX 44 and to address converter 56b via MUX 45. Thus, the cache address is provided to data arrays 51-54 and to tag arrays 61-64. Data arrays 51-52 read out the selected cache line to MUX 57a, and tag arrays 61-62 read out corresponding tag fields to comparator 58a. Comparator 58a compares the tag fields with the tag address received from address converter 56a, and selects which data (if any) MUX 57a forwards to MUX 46. Similarly, data arrays 53-54 read out the selected cache line to MUX 57 b, and tag arrays 63-64 read out corresponding tag fields to comparator 58b. Comparator 58b compares the tag field with the tag address received from address converter 56b, and selects which data (if any) MUX 57b forwards to MUX 46.

The select signals provided by comparators 58a and 58b are compared in comparator 49a to generate a select signal that is provided to MUX 46 via AND gate 48 a to select which data (if any) is returned to CPU 21a. Thus, **if there is matching data in either cache memory partition 23a or cache memory partition 23b, it is returned to CPU 21a via MUX 46. In this manner, data arrays 51-54 provide a 4-way associative cache memory for CPU 21a.** Values for control signals C44-C47 for the 1T and 2T states are summarized below in Table 1.

TABLE 1

mode	C44	C45	C46	C47
1T	0	1	1	d/c
2T	0	0	0	0

As discussed above, ***the ability to easily transition between using cache memory 23 as two dedicated 2-way associative cache memories for respective CPUs 21a-21b, and using cache memory 23 as a 4-way associative memory for only one CPU 21a, advantageously allows for use of the entire cache memory 23, irrespective of whether MPU 20 is executing one or two threads, and thereby maximizes the effectiveness of cache memory 23.*** Further, since CPUs 21a-21b do not simultaneously share access to the same data in cache memory 23, cache controllers 42a and 42b do not need to perform separate chip-level snoop operations, and thus are much simpler and occupy less silicon area than cache controllers for a shared cache memory system.
[Cherabuddi, 6:27–8:9, emphasis added]

(152) In the passages above, Cherabuddi discloses that the system ***monitors past cache performance associated with processors and partitions.*** For instance, Cherabuddi discloses that the caches are *writeback* in design, which means that they monitor whether the processors have modified any of the cache blocks in their partitions (“In response to M=1, dirty data in cache memory partition 23b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23b.”).

(153) In addition, in the passages above, Cherabuddi indicates that ***the operating-system kernel monitors the past performance associated with processors and partitions*** in the system to determine whether or not to repartition the

caches. For instance, Cherabuddi says that, “If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition MPU 20 to the 2T state by changing the mode signal to M=1.”

(154) In the passages above, Cherabuddi also describes how the ***desired partitioning of the system relates the upper level cache block allocations to processors***. When two CPU cores are activated, each is allocated its own cache partition to itself. When only one CPU core is activated, both cache partitions are allocated to that CPU core for its use. The decision to use two cores, and therefore split the cache in two such that the upper level cache blocks are allocated to both processors, is made when, for example, “If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads,” and the decision to use only one core and therefore unify the cache into a monolithic resource such that all upper level cache blocks are allocated to one processor, is made when, for example, “if the kernel determines that only one of CPUs 21a-21b is necessary for a particular instruction code sequence.”

(155) Additionally, in the passages above, Cherabuddi describes ***how the system is repartitioned***, for instance when the resources of the cache memory are

dynamically allocated and reallocated between the first and second CPUs (if the allocation changes, the system has been repartitioned). In particular, Cherabuddi says that the step of repartitioning the system includes allocating the upper level cache blocks to one or more of the CPU cores (e.g., “Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources.”).

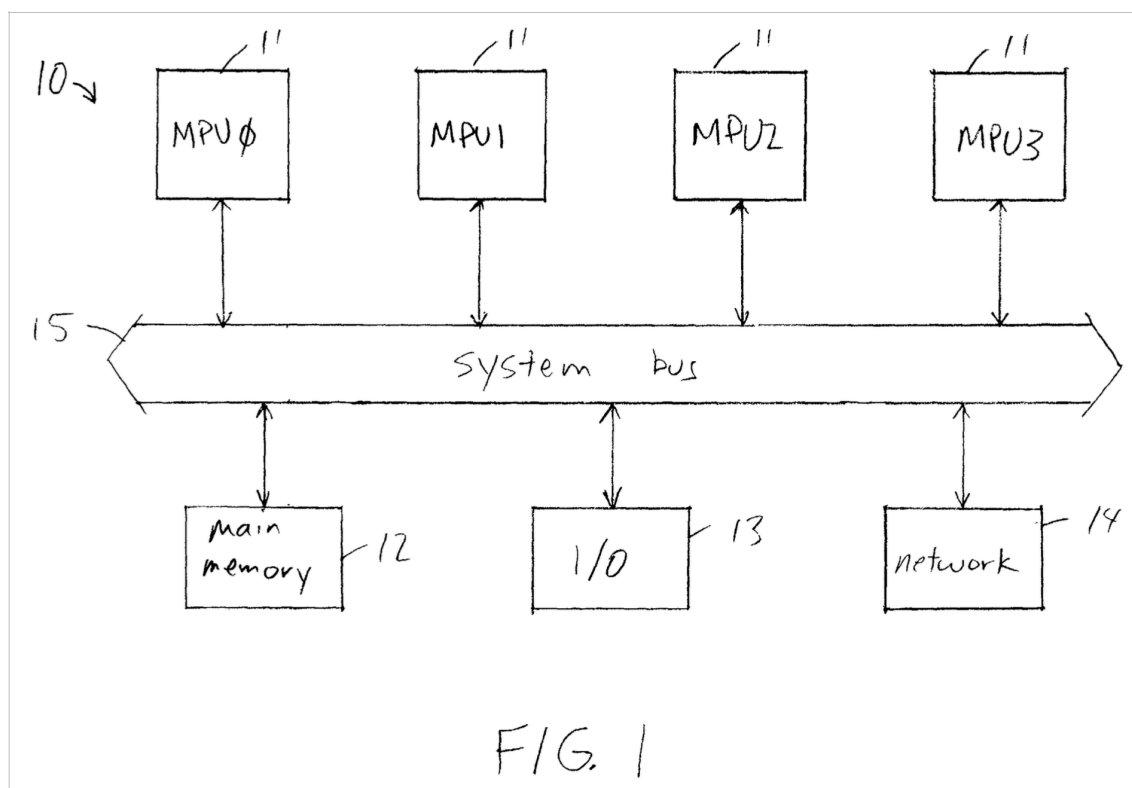
(156) Therefore, Cherabuddi discloses monitoring past cache performance associated with processors and partitions; determining desired processor to partition and upper level cache block allocations to processors; and repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits.

g) wherein the upper level cache blocks are usable to store data and instructions fetched from a random-access main memory.

(157) As described in the *Technical Background* section, caches store data and instructions fetched from another memory level, such as a main memory or

another level of cache. This is what a cache's memory blocks are used for: they house the data and instructions. The operations are performed under the control of a cache controller that does the fetching and, among other functions, keeps the cache consistent and coherent. As illustrated in the *Technical Background* section, split caches hold either instructions or data, and unified caches hold both instructions and data, often just called 'data' because instructions *are* data — they are data words that are interpreted as instructions.

(158) In Figure 1, Cherabuddi discloses that the MPU shown in Figures 2 and 4 above is part of a system with main memory, and that the cache memory blocks store data and instructions fetched from that main memory, a random-access memory such as DRAM:



(159) Cherabuddi describes this as follows:

Some manufactures combine two or more central processing units (CPUs) on a single chip and sell the chip as a multi-processor unit (MPU). The MPU takes advantage of parallel processing to increase performance over a single CPU. ***An MPU typically includes a cache memory to store data in anticipation of future use by the CPUs. The cache memory is smaller and faster than the MPU's main memory, and thus can transfer data to the CPUs in much less time than data from the main memory.*** When data requested by the CPUs is in the cache memory, there is a cache hit, and CPU performance approaches the speed of the cache memory. Conversely, ***when there is a cache miss, the requested data must***

be retrieved from main memory, and thus CPU performance approaches the speed of main memory. Thus, increased performance may be achieved by maximizing the percentage of cache hits during operation. [Cherabuddi, 1:20–27, emphasis added]

FIG. 1 shows a computer system 10 within which embodiments of the present invention may be implemented. System 10 is shown to include four MPUs 11 connected to each other and to a main memory 12, an input/output (I/O) device 13, and a network 14 via a system bus 15. Main memory 12 is shared by MPUs 11, and may be any suitable random access memory (RAM) such as, for example, DRAM. I/O device 13 allows a user to interact with system 10, and may include, for example, a computer monitor, keyboard, and/or mouse input. Network 14 may be any suitable network such as, for example, a local area network, a wide area network, and/or the Internet. Additional devices may be connected to the system bus 15 as desired. [Cherabuddi, 3:13–25, emphasis added]

(160) Cherabuddi's description above confirms the understanding that, in the preceding figure, the MPU shown in Figures 2 and 4 above is part of a system with main memory, and that the cache memory blocks store data and instructions fetched from that main memory, a random-access memory such as DRAM.

(161) Further, a person of ordinary skill would have understood what is described in the *Technical Background* section, which is that caches store data and instructions

fetched from another memory level, such as a main memory or another level of cache. This is what a cache's memory blocks are used for: they house the data and instructions. The operations are performed under the control of a cache controller that does the fetching and, among other functions, keeps the cache consistent and coherent. A person of ordinary skill would have known all of this. A person of ordinary skill would also have understood the concept illustrated in the *Technical Background* section, that split caches hold either instructions or data, and unified caches hold both instructions and data, often just called 'data' because instructions *are* data — they are data words that are interpreted as instructions. Cherabuddi illustrates a unified cache in Figure 4, which would have informed a person of ordinary skill that the cache would hold both instructions and data. Therefore, a person of ordinary skill would have understood that Cherabuddi's cache memory blocks are usable by the cache controllers to store data and instructions fetched from a random-access memory.

(162) It is my opinion that Cherabuddi discloses all the elements of Claim 5 and thus renders the claim obvious.

5. Claim 8

(163) Claim 8 reads as follows:

8. The method of claim 5, wherein the multiple processor integrated circuit further comprises a plurality of non-allocable cache memory blocks.

(164) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

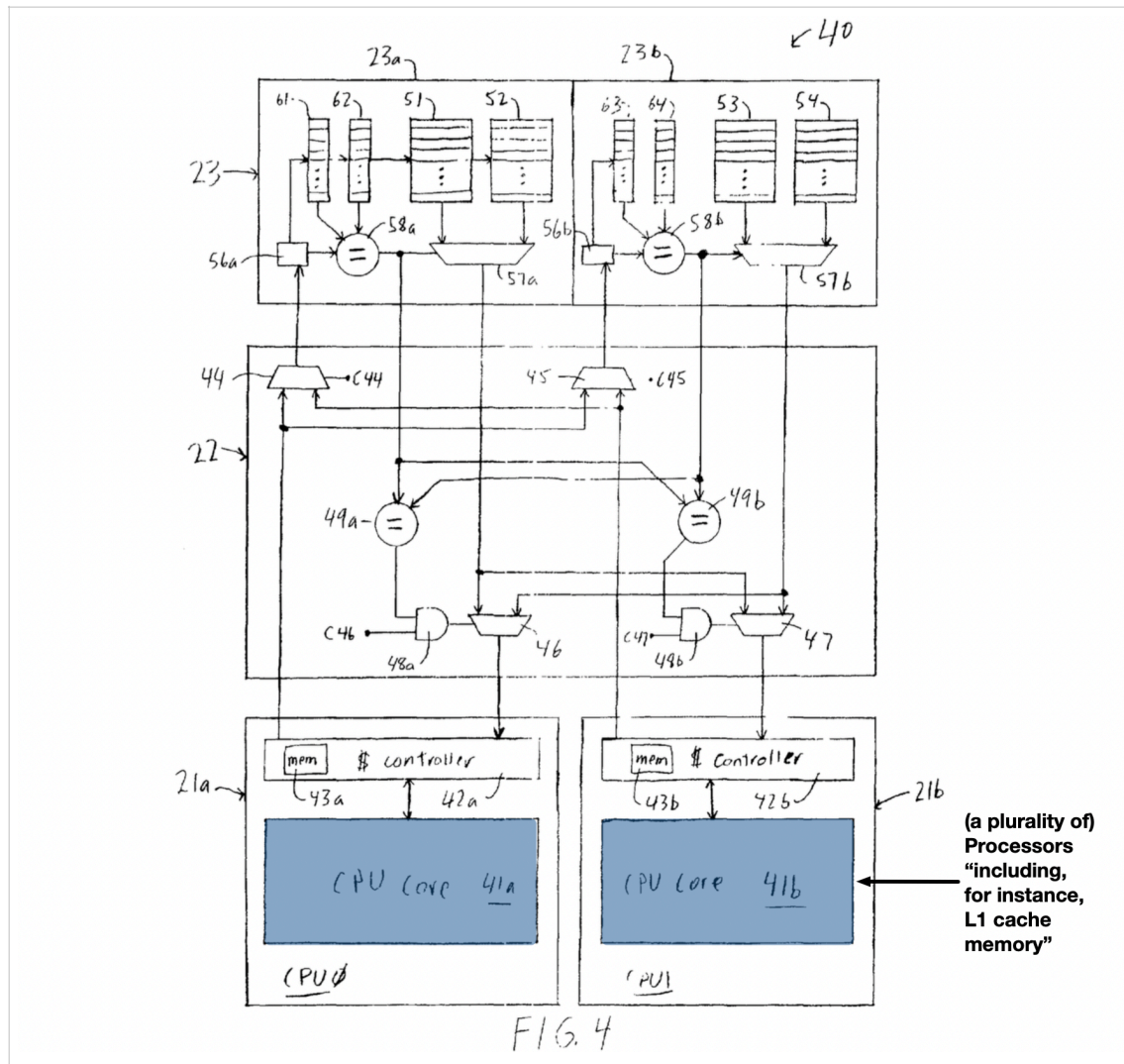
(165) As shown above, the limitations for Claim 5 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein the multiple processor integrated circuit further comprises a plurality of non-allocable cache memory blocks.

(166) Cherabuddi discloses this limitation in two separate ways. First, Cherabuddi discloses that each processor on the multiple processor integrated circuit contains a private L1 cache which, by definition, contains a plurality of blocks that are non-allocable to the other processor. Second, Cherabuddi's assignment of cache memory blocks to processors is done in such a way that, when a block is assigned to a particular processor, it is non-allocable to the other core, at least until a following reassignment/repartitioning occurs.

(167) Regarding the first way in which Cherabuddi discloses this limitation, Cherabuddi discloses that the processors in Figure 4, highlighted below as the CPU

cores 41a and 41b (but which could just as easily be the CPUs 21a and 21b), contain within them L2 caches. The figure is shown below:



(168) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory

partitions 23a-23b. **Each CPU 21 is shown to include a CPU core 41** and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. **CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory**, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

(169) Cherabuddi specifically says that each processor contains “L1 cache memory,” and a person of ordinary skill would have known that L1 caches are private to the processor core¹ and are therefore non-allocable to other processors. Thus, the multiple processor integrated circuit comprises a plurality of non-allocable cache memory blocks.

(170) Regarding the second way in which Cherabuddi discloses this limitation, as Cherabuddi describes, when the processor is in double-core mode, the cache memory blocks assigned to one CPU core are non-allocable to the other core (the other core cannot access those blocks). Cherabuddi describes this as follows:

¹ L1 caches are private to the processor core unless explicitly shown to be shared, which is an unusual configuration that is **not** shown in Figure 4 above, nor is it described in any other figure or discussion in Cherabuddi.

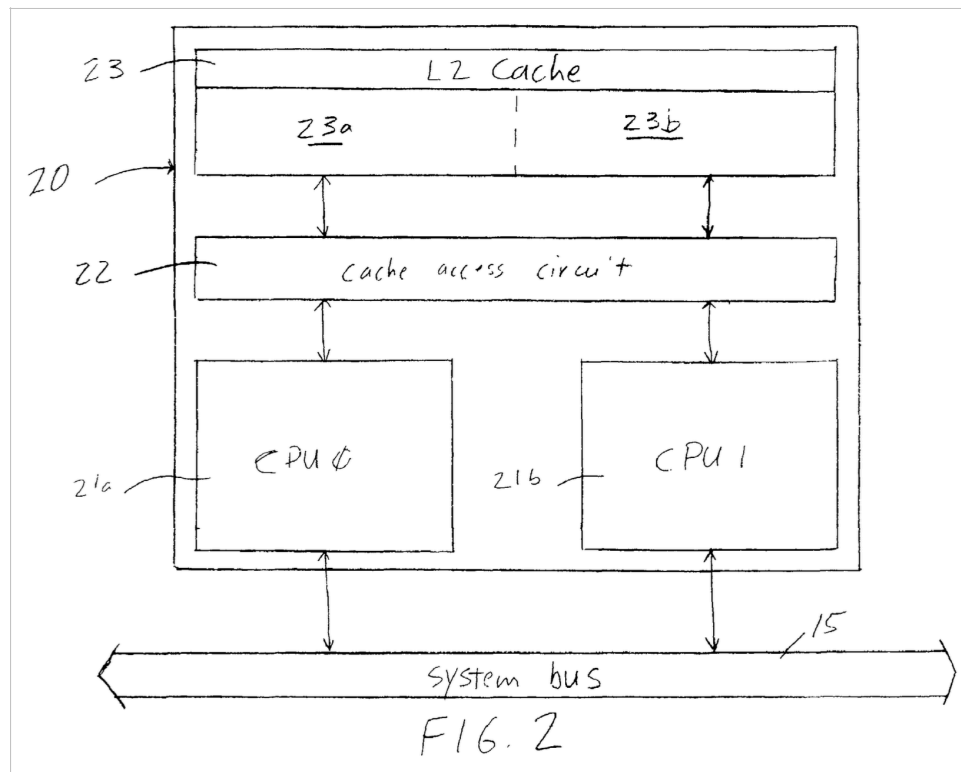
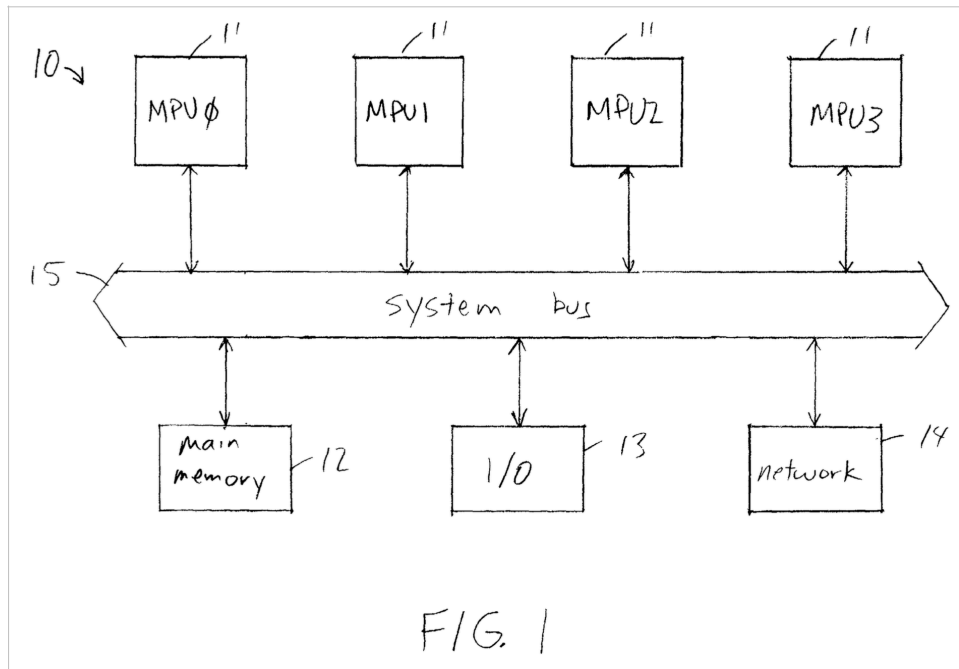


FIG. 2 shows an MPU 20 that is one embodiment of MPU 11 of FIG. 1. MPU 20 is shown to include first and second processors such as central processing units (CPUs) 21a-21b, a cache access circuit 22, and a dynamically allocated cache memory 23.

CPUs 21a-21b are well-known processing devices. **Cache memory 23 is partitioned into first and second cache memory partitions 23a-23b, and is preferably a high speed cache memory device such as SRAM,** although other cache devices may be used. For the purpose of discussion herein, each cache memory partition 23a-23b is configured as a 2-way associative cache memory. Of course, in actual embodiments, the cache memory partitions may be configured for other levels of associativity.

Cache access circuit 22 selectively couples the first and/or second CPUs 21a-21b to the first and/or second cache memory

partitions 23a-23b. As explained in detail below, cache access circuit 22 allows the resources of cache memory 23 to be dynamically allocated between the first and second CPUs 21a-21b according to each CPU's processing requirements to more efficiently utilize cache resources.



Referring also to FIG. 1, *system 10 includes well-known system operating software that assigns tasks of one or more computer programs running thereon to the various MPUs 20 for execution. The operating software, which is often referred to as the system kernel, also assigns tasks between the CPUs 21a-21b of each MPU 20. For applications that include a single instruction execution thread and are thus best executed using only one CPU 21, e.g., for applications having a highly sequential instruction code, the kernel assigns all the tasks to one CPU and idles the other CPU. Conversely, for applications that can be divided into two parallel instruction execution*

threads, e.g., for applications having parallel execution loops, the kernel may assign different threads to CPUs 21a-21b for simultaneous execution therein.

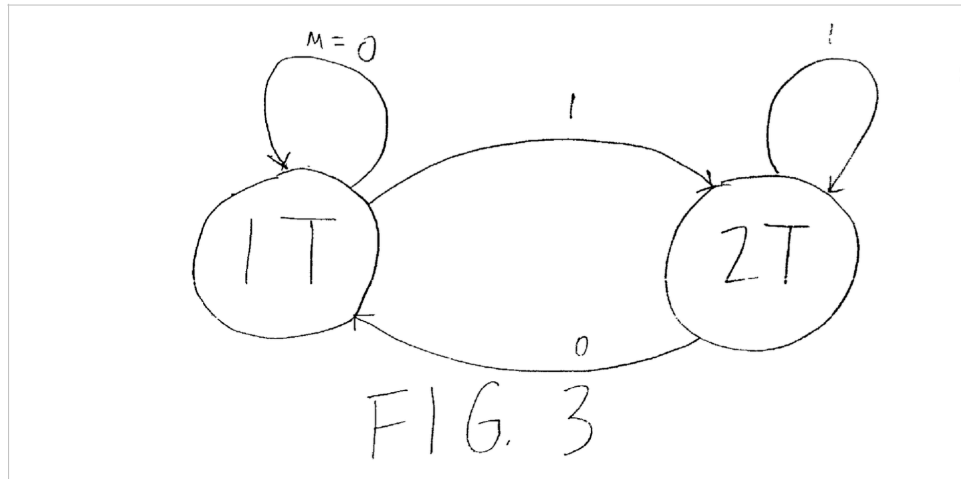


FIG. 3 illustrates state transitions of MPU 20 between a one-thread (1T) state and a two-thread (2T) state. In one embodiment, upon power-up of MPU 20, the kernel sets a mode signal $M=0$ to initialize MPU 20 to the 1T state. The kernel sets one of the CPUs 21 to an active state and sets the other CPU 21 to an inactive state. For purposes of discussion herein, ***during the 1T state, the kernel sets CPU 21a as the active CPU and sets CPU 21b as the inactive CPU***, although in other embodiments the kernel may set CPU 21b as the active CPU and set CPU 21a as the inactive CPU. ***While in the 1T state, the kernel assigns tasks of the computer program(s) only to the active CPU 21a, while the other CPU 21b remains idle. In response to $M=0$, cache access circuit 22 couples the first CPU 21a to both the first and second cache memory partitions 23a-23b to allow the first CPU 21a to use all resources of cache memory 23. In this state, the active CPU 21a may use cache memory partitions 23a-23b as a 4-way associative cache memory.***

If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21a-21b to the active state, and thereafter assigns one execution thread to CPU 21a and another execution thread to CPU 21b in a well-known manner. In response to M=1, dirty data in cache memory partition 23b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23b. *The cache access circuit 22 couples the first CPU 21a to the first cache memory partition 23a for exclusive access thereto, and couples the second CPU 21b to the second cache memory partition 23b for exclusive access thereto. In this state, CPU 21a may use cache memory partition 23a as a dedicated 2-way associative cache memory, and CPU 21b may use cache memory partition 23b as a dedicated 2-way associative cache memory.*

Thereafter, if the kernel determines that only one of CPUs 21a-21b is necessary for a particular instruction code sequence, the kernel may transition MPU 20 to the 1T state by changing the mode signal to M=0, flushing the second cache memory partition 23b, and then assigning execution of the instruction code sequence to the active CPU 21a. [Cherabuddi, 3:26–4:37, emphasis added]

(171) Cherabuddi's description above confirms the understanding that, when the processor is in double-core mode, the cache memory blocks assigned to one CPU core are non-allocable to the other core (the other core cannot access those blocks).

For instance, the passage above explains that, “When M=1, the kernel sets both CPUs 21a-21b to the active state ... The cache access circuit 22 couples the first CPU 21a to the first cache memory partition 23a ***for exclusive access thereto***, and couples the second CPU 21b to the second cache memory partition 23b ***for exclusive access thereto***.” In this state, there is a plurality of non-allocable cache memory blocks: as Cherabuddi makes clear, the cache memory blocks assigned to one CPU core are non-allocable to the other core (the other core cannot access those blocks).

(172) It is my opinion that Cherabuddi discloses all the elements of Claim 8 and thus renders the claim obvious.

6. Claim 9

(173) Claim 9 reads as follows:

9. The method of claim 5, wherein the interconnect apparatus further comprises a plurality of upper level cache controllers, and where each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache.

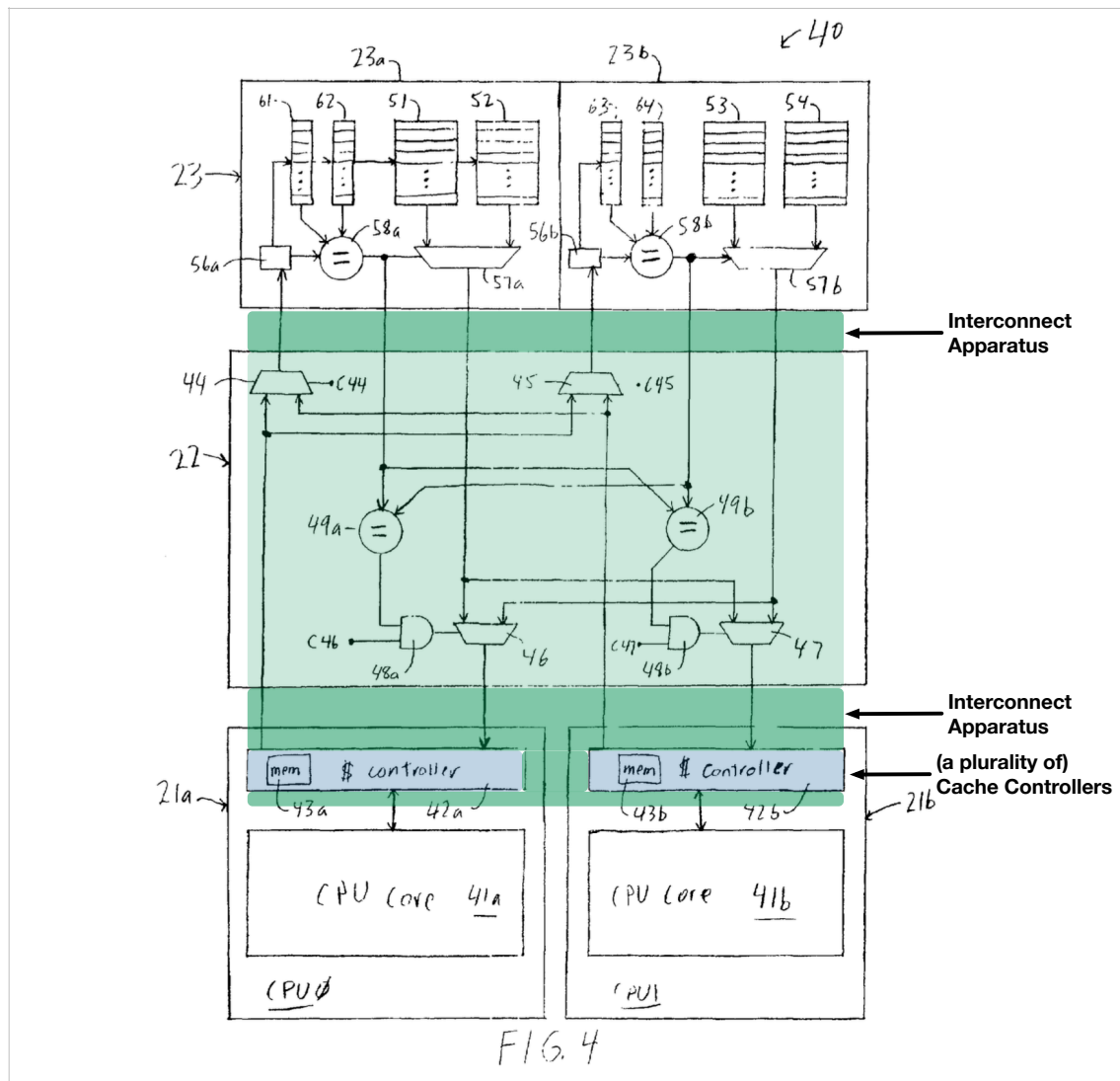
(174) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

(175) As shown above, the limitations for Claim 5 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein the interconnect apparatus further comprises a plurality of upper level cache controllers,

(176) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, the plurality of cache memory banks is at the top (elements 23a and 23b), the cache controllers are highlighted in light blue, and the interconnect apparatus is highlighted in green. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b, which are connected to the cache controllers through the wires in the interconnect apparatus.



(177) Cherabuddi describes the controllers as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. **Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and**

data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on. [Cherabuddi, 4:66–5:14, emphasis added]

(178) Cherabuddi's description above confirms the understanding that, in the preceding figure, the plurality of cache memory banks is at the top (elements 23a and 23b), the cache controllers are highlighted in light blue, and the interconnect apparatus is highlighted in green. In the figure is shown a set of level-2 cache memory blocks in each of the L2 Cache partitions 23a and 23b, which are connected to the cache controllers through the wires in the interconnect apparatus.

c) and where each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache.

(179) Cherabuddi discloses that each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache, by disclosing that the caches are indeed writeback:

If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21a-21b to the active state, and thereafter assigns one execution thread to CPU 21a and another execution thread to CPU 21b in a well-known manner. ***In response to M=1, dirty data in cache memory partition 23b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23b.*** The cache access circuit 22 couples the first CPU 21a to the first cache memory partition 23a for exclusive access thereto, and couples the second CPU 21b to the second cache memory partition 23b for exclusive access thereto. In this state, CPU 21a may use cache memory partition 23a as a dedicated 2-way associative cache memory, and CPU 21b may use cache memory partition 23b as a dedicated 2-way associative cache memory. [Cherabuddi, 3:13–31, emphasis added]

(180) Cherabuddi's description above confirms the understanding that the caches are writeback. For instance, in the passage above, Cherabuddi discloses, "In response to M=1, ***dirty data in cache memory partition 23b is written back to main memory 12 using a well-known writeback operation,*** thereby flushing cache memory partition 23b."

(181) It is my opinion that Cherabuddi discloses all the elements of Claim 9 and thus renders the claim obvious.

7. Claim 10

(182) Claim 10 reads as follows:

10. The method of claim 5, wherein each upper level allocable cache block further comprises tag memory and cache data memory.

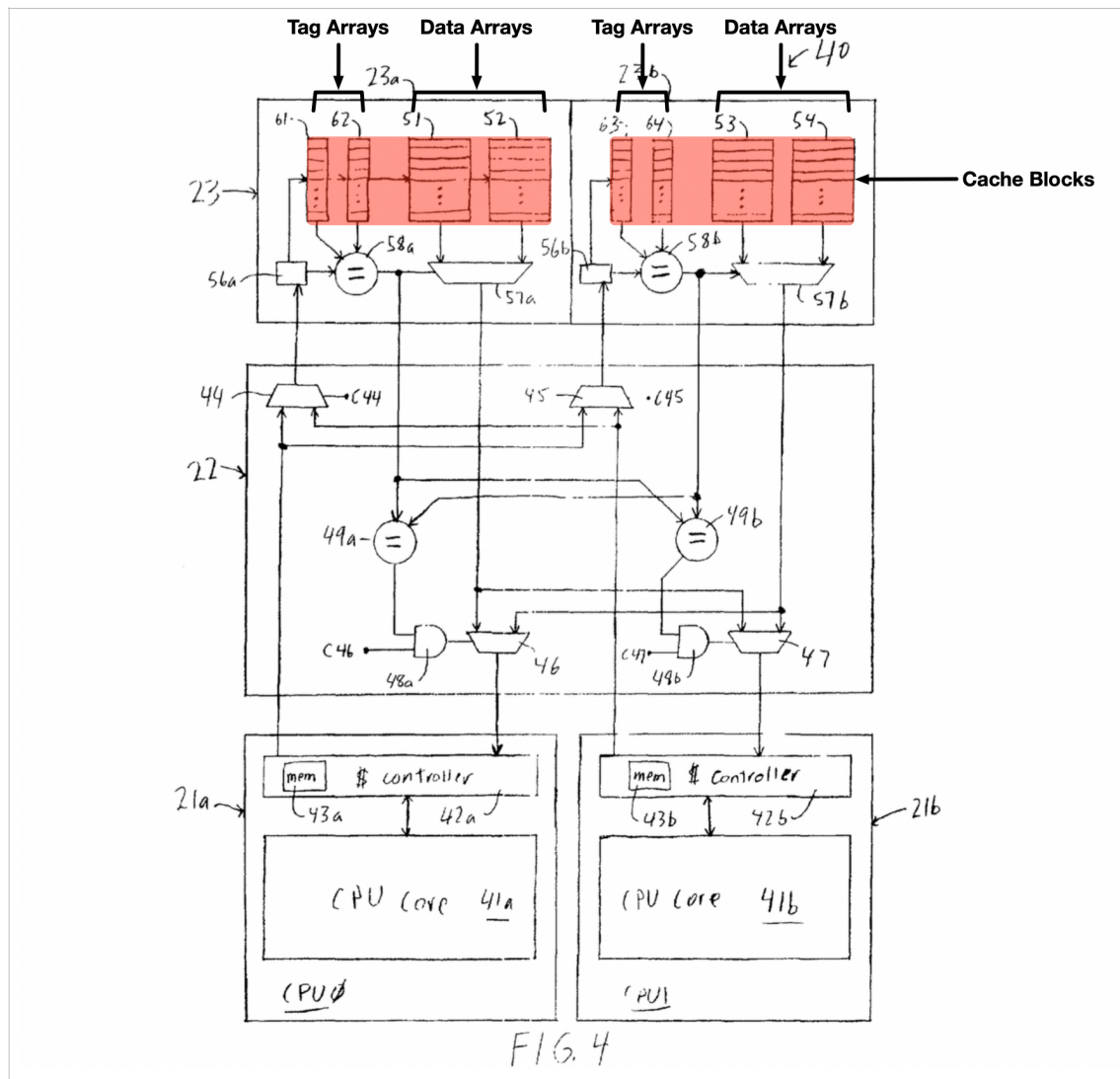
(183) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

(184) As shown above, the limitations for Claim 5 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein each upper level allocable cache block further comprises tag memory and cache data memory.

(185) Cherabuddi discloses this limitation in Figure 4, shown below; in the figure, each upper level allocable cache block is highlighted in red. In the figure is shown a set of level-2 cache memory blocks that comprise cache tag memory. The tag memory blocks are labeled as elements 61 and 62 in partition 23a (left side), and elements 63 and 64 in partition 23b (right side).



(186) Cherabuddi describes this as follows:

FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23a-23b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data

between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.[Cherabuddi, 4:66–5:14, emphasis added]

Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23b includes a well-known address converter 56b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54.

Data arrays 51-54 each include a plurality of cache lines for storing data retrieved from main memory 12. Cache lines in data arrays 51-54 may be any suitable length. In one embodiment, each cache line of data arrays 51-54 stores 32 Bytes of data. Each data array 51-54 also includes a well-known address decoder (not shown for simplicity) that selects a cache line for read and write operations in response to a received cache index. Data arrays 51-52 provide data at a selected

cache line to a MUX 57a, and data arrays 53-54 provide data at selected cache line to a MUX 57b.

Tag arrays 61-64 each include a plurality of lines for storing tag information for corresponding cache lines in data arrays 51-54, respectively. Tag arrays 61-62 provide tags at the selected cache line to a comparator 58a which, in response to a comparison with a tag address received from address converter 56a, generates a select signal for MUX 57a. Similarly, tag arrays 62 and 63 provide tags at the selected cache line to a comparator 58b which, in response to a comparison with a tag address received from address converter 56b, generates a select signal for MUX 57b. Comparators 58a and 58b are well-known. [Cherabuddi, 5:15–49, emphasis added]

(187) Cherabuddi's description above confirms the understanding that, in the preceding figure, each upper level allocable cache block is highlighted in red. In the figure is shown a set of level-2 cache memory blocks that comprise cache tag memory. The tag memory blocks are labeled as elements 61 and 62 in partition 23a (left side), and elements 63 and 64 in partition 23b (right side). Cherabuddi explicitly describes the elements as tag-memory arrays in the passage above: "Cache memory partition 23a includes two data RAM arrays 51-52 having corresponding searchable ***tag arrays*** 61-62, respectively, while cache memory partition 23b includes two data RAM arrays 53-54 having corresponding searchable ***tag arrays*** 63-64, respectively." Therefore, the cache memory blocks (elements highlighted in red in the figure above) comprise cache tag memory.

(188) It is my opinion that Cherabuddi discloses all the elements of Claim 10 and thus renders the claim obvious.

B. The Combination of Cherabuddi and Dean

(189) In my opinion, it would have been obvious for a person of ordinary skill to combine Cherabuddi and Dean. The choice would have been obvious, and the steps to make it happen would have been obvious, as Cherabuddi and Dean are natural complements to one another. Both Cherabuddi and Dean describe a multiprocessor system in which the processors, coupled together, share a level of cache memory in a way that portions of the cache memory can be assigned to certain processors and therefore be dedicated to them. While both Cherabuddi and Dean state that the point of partitioning the cache memory space is to be done to optimize the system's performance, Cherabuddi leaves the details of the partitioning algorithm to the designer (e.g., Cherabuddi says that "the resources of a partitioned cache memory are dynamically allocated between two or more processors on a multi-processor unit [MPU] according to a desired system configuration or to the processing needs of the processors" [Cherabuddi, 1:62–66]), whereas Dean describes in detail how the partitioning can be chosen. The two mechanisms are complementary, and their combination would yield a simple and effective design that optimizes the system's performance. Cherabuddi and Dean are

both reasonably pertinent to a problem described by the '264 Patent, that of improving performance.

(190) Cherabuddi describes the multiprocessor-based cache system as a merging of two processors and a cache that can be divided between the two processors:

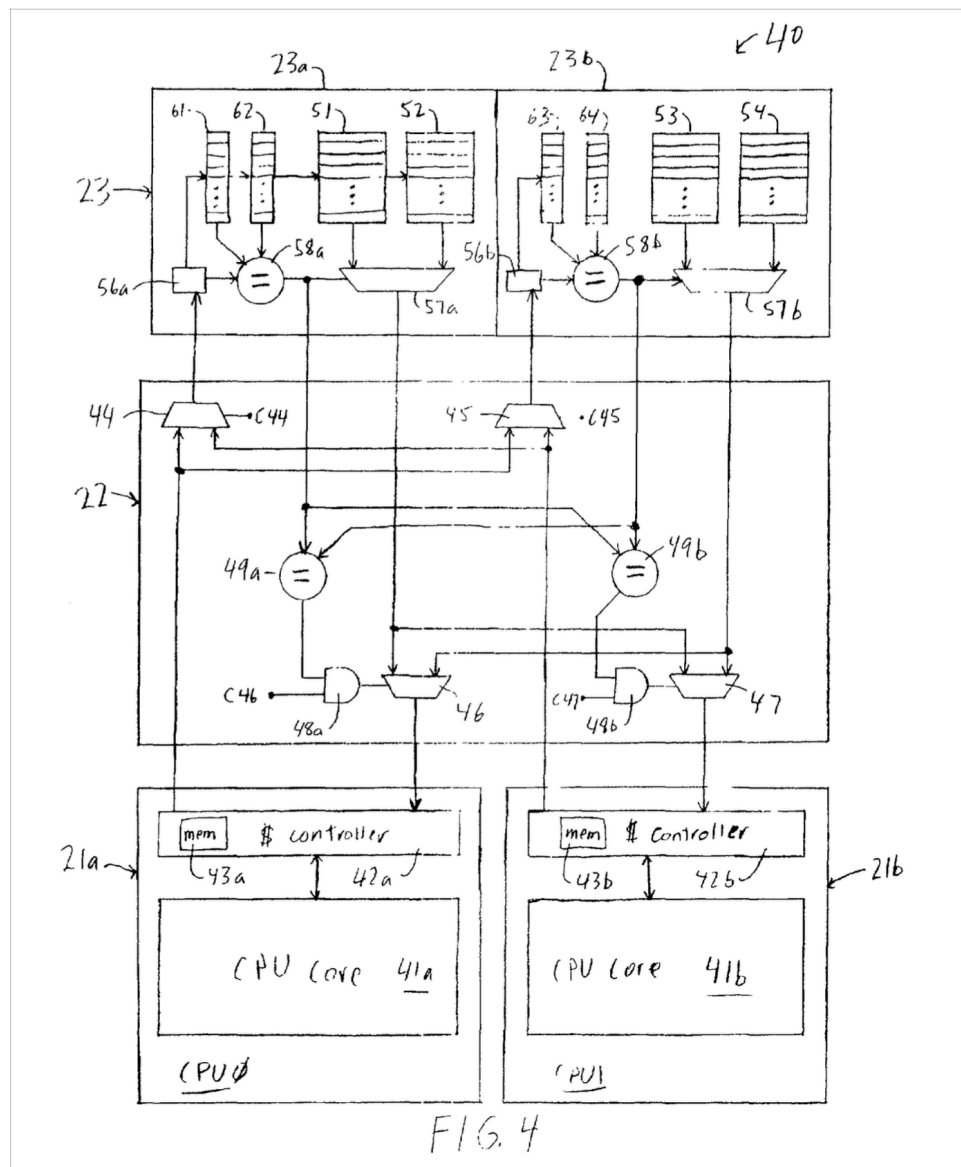


FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23 a-23 b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.

Cache memory partition 23 a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23 b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23 a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23 b includes a well-known address converter 56 b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54. [Cherabuddi, 4:66–5:28]

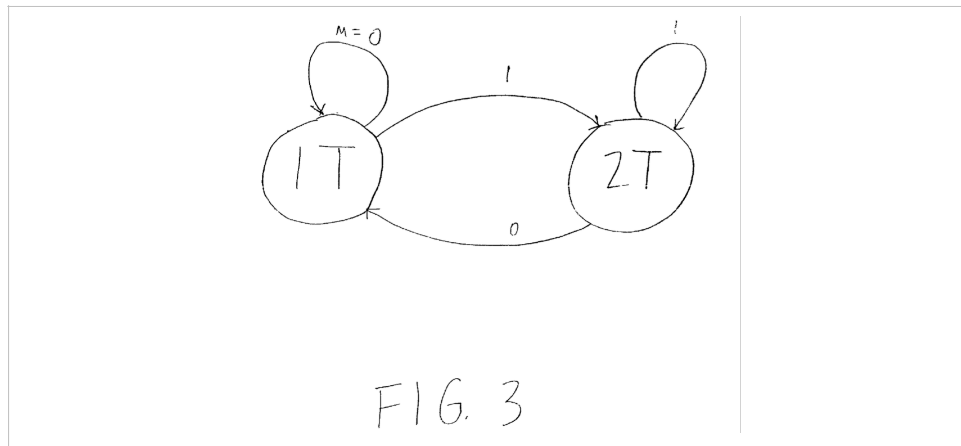


FIG. 3 illustrates state transitions of MPU 20 between a one-thread (1T) state and a two-thread (2T) state. In one embodiment, upon power-up of MPU 20, **the kernel sets a mode signal $M=0$ to initialize MPU 20 to the 1T state. The kernel sets one of the CPUs 21 to an active state and sets the other CPU 21 to an inactive state.** For purposes of discussion herein, during the 1T state, the kernel sets CPU 21 a as the active CPU and sets CPU 21 b as the inactive CPU, although in other embodiments the kernel may set CPU 21 b as the active CPU and set CPU 21 a as the inactive CPU. **While in the 1T state, the kernel assigns tasks of the computer program(s) only to the active CPU 21 a, while the other CPU 21 b remains idle. In response to $M=0$, cache access circuit 22 couples the first CPU 21 a to both the first and second cache memory partitions 23 a-23 b to allow the first CPU 21 a to use all resources of cache memory 23. In this state, the active CPU 21 a may use cache memory partitions 23 a-23 b as a 4-way associative cache memory.**

If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition

MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21 a-21 b to the active state, and thereafter assigns one execution thread to CPU 21 a and another execution thread to CPU 21 b in a well-known manner. In response to M=1, dirty data in cache memory partition 23 b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23 b. The cache access circuit 22 couples the first CPU 21 a to the first cache memory partition 23 a for exclusive access thereto, and couples the second CPU 21 b to the second cache memory partition 23 b for exclusive access thereto. ***In this state, CPU 21 a may use cache memory partition 23 a as a dedicated 2-way associative cache memory, and CPU 21 b may use cache memory partition 23 b as a dedicated 2-way associative cache memory.***

Thereafter, if the kernel determines that only one of CPUs 21 a-21 b is necessary for a particular instruction code sequence, the kernel may transition MPU 20 to the 1T state by changing the mode signal to M=0, flushing the second cache memory partition 23 b, and then assigning execution of the instruction code sequence to the active CPU 21 a. [Cherabuddi, 3:61–4:37, emphasis added]

(191) Cherabuddi also suggests partitioning cache memory dynamically allocated between two or more processors on an MPU based on “a desired system configuration or to the processing needs of the processors.” [Cherabuddi, 1:61–66] In particular, a cache memory is partitioned between two or more processors and treats each cache memory partition as a dedicated cache memory. [Cherabuddi,

2:8–12] In this scenario, the entire cache memory can dynamically allocate cache resources between multiple processors to maximize cache resources while allowing each processor and corresponding allocated cache memory partition to operate independently. [Cherabuddi, 2:39–44] Cherabuddi’s invention also allows for any processor to access the entire cache memory in case another processor is found to be defective, maximizing cache resources. In the passage below, Cherabuddi says that the partitioning decision-making is done “in response to specific needs of associated CPUs,” thereby “ensuring that both cache memory partitions 23 a-23 b are utilized, irrespective of whether one or both CPUs 21a-21 b are active.”

By dynamically allocating resources of cache memory 23 in response to specific needs of associated CPUs 21 a-21 b, embodiments of the present invention maximize cache performance by ensuring that both cache memory partitions 23 a-23 b are utilized, irrespective of whether one or both CPUs 21a-21 b are active. Thus, in the 1T state, both cache memory partitions 23 a-23 b are allocated to the active CPU, and in the 2T state, each cache memory partition 23 a and 23 b is allocated only to its corresponding CPU 21 a and 21 b, respectively. Since allocation of cache memory partitions 23 a-23 b is controlled by cache access circuit 22, cache memory 23 does not require any special hardware, and thus may be of conventional architecture. Further, since cache memory 23 is not shared between CPUs 21 a-21 b, all snoop operations may be performed at the system level. As a result, the cache controllers (not shown in FIG. 2) in CPUs 21 a-21 b are much simpler

and occupy less silicon area than cache controllers for shared cache memory systems. [Cherabuddi, 4:38–55, emphasis added]

(192) However, Cherabuddi does not provide details on any specific partitioning algorithm. Cherabuddi observes that “increased performance may be achieved by maximizing the percentage of cache hits during operation.” [Cherabuddi, 1:26–27] Therefore, a person of ordinary skill designing a system based on Cherabuddi would have been motivated to develop the algorithm to be used to partition the system’s cache-memory resources, increasing performance by maximizing the percentage of cache hits during operation. That person would have found it obvious to seek out references providing description of a partitionable cache system in which the partitioning algorithm is given, and specifically a reference showing how to achieve increased performance by maximizing the percentage of cache hits during operation.

(193) Dean is such a reference: Dean provides a description of a partitionable cache system in which the partitioning algorithm is given, specifically showing how to achieve increased performance by maximizing the percentage of cache hits during operation. Dean teaches that “[t]he system and method preferably determine[] which process could benefit the most if additional cache space was allocated to the process or could benefit a computer system the most if additional cache space was allocated to the process.” [Dean, 3:58–62] Dean also teaches that

“[t]he reallocation can take place periodically, thereby maximizing the performance of the system and handling unforeseen demands by processes.”

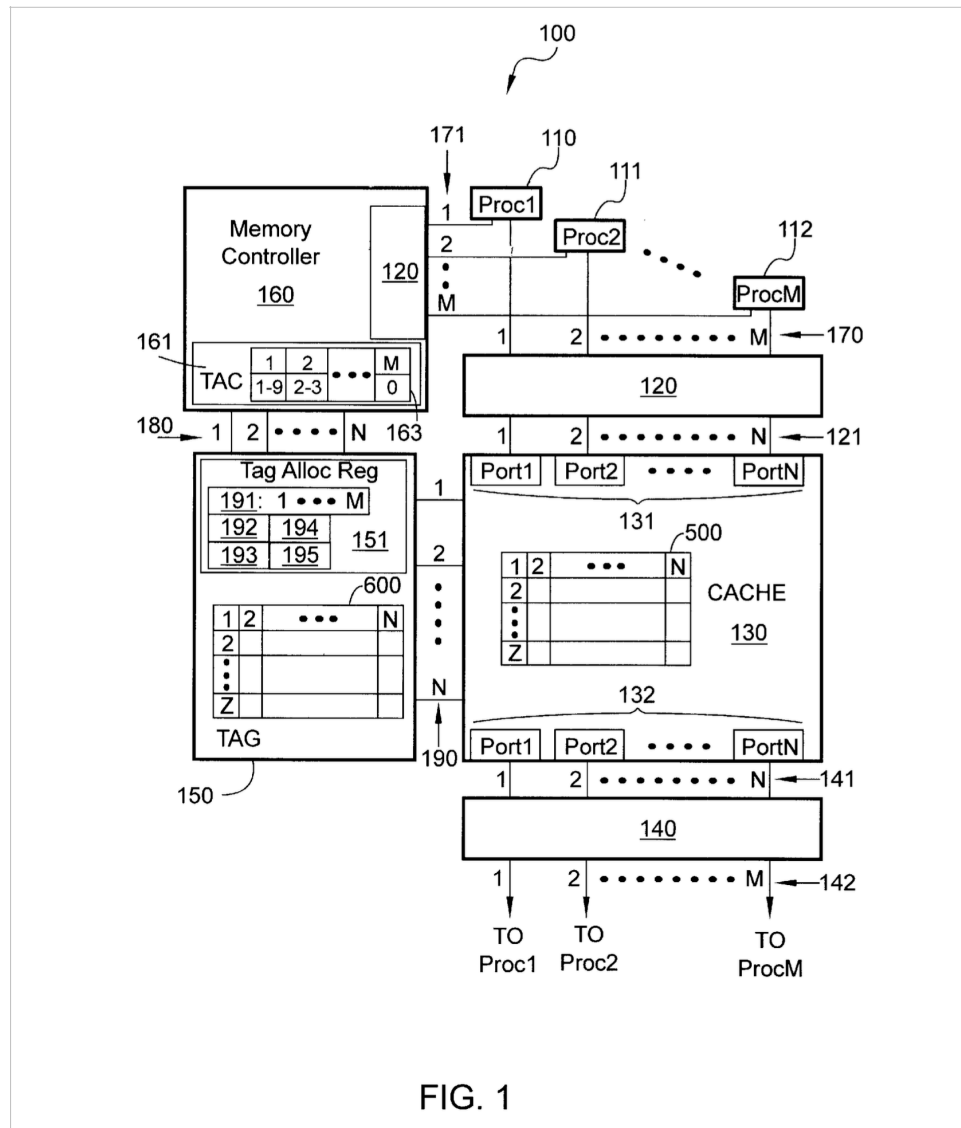
[Dean, 3:64–66] Dean explains that “a relational cache miss percentage is used to reallocate the ways, but other system metrics may also be used.” [Dean, 3:9–11]

Accordingly, a person of ordinary skill would have been motivated to improve system performance by dynamically reallocating cache memory as taught in Cherabuddi by monitoring cache hit rates as taught in Dean.

(194) Dean describes a multiprocessor cache that is partitionable between different processors. Dean’s Abstract is given below, which says that the cache miss rate is to be used in the partitioning algorithm:

The present invention provides a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and multiple single-cycle cache accesses from a multiport cache. ***Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways***, but other metrics may also be used. [Dean, Abstract, emphasis added]

(195) Dean's cache design is shown below:



Referring now to FIG. 1, a system 100 is shown that provides a multiport cache and tag structure and allows dynamic allocation of cache ways. System 100 comprises M processes 110, 111, 112, a processes-to-cache multiplexor 120, a cache-to-processes multiplexor 140, a multiport cache 130, a memory controller 160, and a tag 150.

Memory controller 160 comprises a processes-to-cache multiplexor 120, N output addresses 180, and a tag allocation controller 161. Processes-to-cache multiplexor 120 in memory controller 160 has N outputs (not shown) that are routed to the N addresses 180. Processes 110, 111, 112 can be individual processors that access the memory controller and cache through buses 170, 171 or can be tasks that are running on one processor and accessing the memory controller and cache through buses 170, 171. These processes communicate the address of the data to the memory controller through bus 171, write data through bus 170, and read data through bus 142. [Dean, 4:6–23]

(196) Dean’s cache can handle not just multiple CPU cores but “multiple processes, which could be processors, tasks, or threads.” [Dean, 3:50–52]

(197) Dean describes the cache-partitioning approach in detail. For instance, Dean says the following:

Cache miss percentage 194 is a factor used to determine if the cache should be reallocated. The total misses for all of the process are compared to the cache miss percentage to determine if there are too many misses. The cache miss percentage 194 thus reduces the number of times the cache is reallocated. The cache miss percentage 194 is optional, but preferred. The cache reallocation factor 195 is a factor used to reallocate the cache, and it determines what percentage of the cache will be reallocated to a task. This factor will generally be a certain number of ways of the cache or a certain percentage of the cache. [Dean, 3:45–4:23, emphasis added]

When the cache evaluation period has completed (step 255=YES), it is first determined if the total number of cache misses is large enough to warrant a cache reallocation cycle (step 260). This is done by adding miss counters A, B, C, and D in system metric 191 and comparing the result to a cache miss percentage 195. The cache miss percentage 195 can be calculated by multiplying the expected or desired cache hit ratio for each process by the cache evaluation period and adding up the results. ***If the total number of misses is less than the cache miss percentage, this indicates that the current cache allocation is working and the current cache allocation should be left alone.*** In this event (step 260=NO), the counters are reset for the current cache cycle counter 192 and the system metric counter(s) 191 (step 265), and another cache evaluation period is started (step 220).

If the sum of the cache miss counters in system metric 191 is greater than the cache miss percentage (step 260=YES), step 270 is performed. Method 270 is shown in FIG. 4, and begins when it is decided if the difference between process cache miss counters is enough to warrant a reallocation cycle (step 271).

If, for example, there are a lot of cache misses, uniformly distributed among all the processes, then the current allocation is considered fair and no reallocation is done. Preferably, the cache is only reallocated when there is a relatively significant disparity in cache miss ratios between processes. Because of this, each cache miss counter for each processor in system metric 191 is compared to the others (step 22). If, for example, process A's miss counter is larger than processor B's miss counter by a pre-determined cache reallocation factor 195, some ways of the cache will be assigned to processor A (step 273). Preferably, this is done by determining which

process has the largest percentage of the cache and taking cache ways from this process (step 273).

Alternatively, the method could determine which process had the least misses during the current cache evaluation cycle and then take cache ways assigned to this process. For the latter method, there may be a minimum number of cache ways that will also factor into this analysis: if the method determines that the process with the lowest number of cache hits currently is assigned the minimum number of ways, the method will leave the cache allocation for this process alone.

The pre-determined cache reallocation factor is a number that indicates how much greater the misses for one process must be versus another process for the method to allocate cache ways to the first process. This factor is preferable to prevent the method from continually reassigning the ways of the cache for small differences in miss rates between the processes. [Dean, 9:27–10:10, emphasis added]

The preferred embodiments of the present invention provide a performance based system and method for dynamic allocation of a unified multiport cache. [Dean,2:66–3:1]

Preferably the process that has its ways assigned to process A would be the process that has the lowest cache miss percentage. This should provide the best increase in performance, as after reallocation process A should have a smaller cache miss percentage and the other process should have a slightly higher cache miss percentage. [Dean,10:35–40]

(198) Cherabuddi describes a system with multiple cores and a cache that can be partitioned among them and says that partitioning should be done “in response to specific needs of associated CPUs [to] maximize cache performance.”

[Cherabuddi, 4:39–41] Dean describes a multi-ported cache that can support multiple cores, and also describes a specific cache-partitioning algorithm based on miss-rate metrics, with the goal to have cache misses “uniformly distributed among all the processes” [Dean, 9:49–50] Dean states in its “Technical Field” that the goal of the system is to achieve high performance:

This invention generally relates to cache circuits, and more specifically relates to a ***performance based system*** and method for dynamic allocation of a unified multiport cache. [Dean, *Technical Field*]

(199) Dean’s cache-miss counters are held in close proximity to the cache’s tag arrays; therefore, in the combined system, the cache-miss counters would be held in close proximity to the cache’s tag arrays (e.g., Cherabuddi’s tag arrays 61, 62, 63, and 64). This means that each of the cache controllers 43a and 43b would be able to see the cache-miss counters.

(200) The modification of combining Dean with Cherabuddi would have been obvious to a person of ordinary skill for several reasons. First, this modification combines familiar elements (caches, cache partitions, the ability to assign different

partitions to different processors, and algorithms to guide the assignment) according to known methods that would yield predictable results. Second, this modification is the addition of one known element (Dean's cache-partitioning algorithm based on miss-rate metrics) into another mechanism (Cherabuddi's partitionable cache that can be partitioned according to a given algorithm) that yields predictable results. Third, it is the use of a known technique (using an algorithm to partition a cache) to improve similar items or methods (Cherabuddi's partitionable cache) that yields predictable results. Fourth, it is the application of a known technique (using miss-rate metrics to decide how to assign cache partitions) to a prior art reference (Cherabuddi) that is ready for improvement that yields predictable results. Fifth, the need to partition the cache, already acknowledged in Cherabuddi, is furthered by the use of a cache-partitioning algorithm based on miss-rate metrics as taught in Dean.

(201) Accordingly, it is my opinion that a person of ordinary skill would have found it obvious and straightforward to combine Cherabuddi and Dean. They are in the same field (they both describe multiprocessor cache systems); they use similar techniques (they both describe a multiprocessor cache in which the cache's memory can be partitioned between the different processors); they are compatible and easily combined (Dean simply describes a specific partitioning algorithm in more detail than Cherabuddi); and their combination produces nothing more than the

expected behavior, which is the sum of both systems. Doing so would have been well within the skill set of the person of ordinary skill.

(202) In that same vein, Cherabuddi also describes the benefit of dynamically allocating cache resources between multiple processors, which maximizes cache resources, similarly increasing performance. Cherabuddi, 2:40-46 (“The ability to dynamically allocate cache resources between multiple processors advantageously allows the entire cache memory to be used, irrespective of whether one or both processors are currently active, thereby maximizing cache resources while allowing for both one-thread and two-thread execution modes. In addition, the present invention may be used to maximize cache resources when one of the on-board processors is defective.”). Likewise, Dean observes that computer system innovation is motivated by the demand for consumers to have faster and more powerful computers. Dean, 1:11-13. A contributing inhibitor is “the speed by which data can be accessed from memory.” Dean, 1:14-17. Dean explains that a multiprocessor may be delayed in processing speed by accessing data from a slow main memory, but cache memory (a small amount of very fast and expensive memory that is used to store a copy of frequently accessed information) can be combined with main memory to increase memory access while remaining a relatively lower cost. Dean, 1:27-30. A person of skill would recognize that the cache miss percentage, monitoring the expected or desired cache hit ratio, as taught

by Dean, would be another metric used in dynamically apportioning cache partitions among the processors in Cherabuddi.

C. Claims 4–5 and 8–10 Are Rendered Obvious by the Combination of Cherabuddi and Dean

(203) In my opinion, the combination of Cherabuddi and Dean renders obvious Claims 4–5 and 8–10 of the '264 patent. My analysis, which includes my reasoning and the bases for my opinion, is provided below.

1. Claim 4

(204) Claim 4 reads as follows:

4. The processor integrated circuit of claim 1, wherein each cache controller is provided with cache hit rate monitoring apparatus.

(205) The following sections analyze this claim in light of the prior art.

a) The processor integrated circuit of claim 1,

(206) As shown above, the limitations for Claim 1 are rendered obvious by Cherabuddi, the analysis and evidence for which is incorporated here by reference.

b) wherein each cache controller is provided with cache hit rate monitoring apparatus.

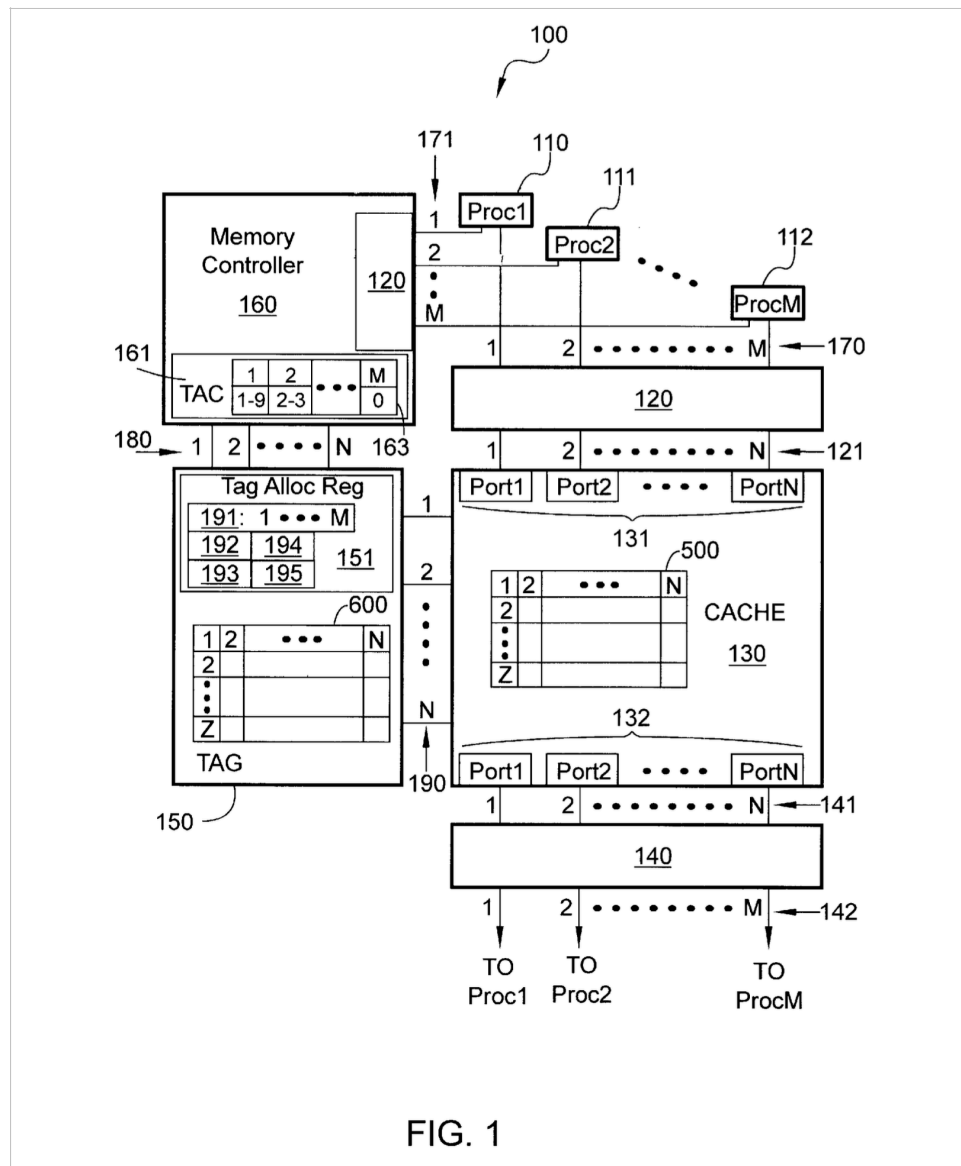
(207) The combination of Cherabuddi and Dean discloses this limitation. As described previously, Dean's system uses a cache-miss percentage to allocate storage in the cache to different processors, and a cache miss rate is simply the inverse of the cache hit rate ($miss_rate = 1 - hit_rate$). This is basic knowledge that a person of ordinary skill would have known. Dean describes the use of the cache-miss percentage below:

The present invention provides a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and multiple single-cycle cache accesses from a multiport cache. ***Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways***, but other metrics may also be used. [Dean, *Abstract*, emphasis added]

The preferred embodiments of the present invention provide a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and

multiple single-cycle cache accesses from a multiport cache. Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. ***Most preferably, a relational cache miss percentage is used to reallocate the ways, but other system metrics may also be used.*** [Dean, 3:45–57, emphasis added]

(208) In the passage above, Dean confirms the understanding that the “relational cache miss percentage” can be used to reallocate, to different processors, the ways of the cache (e.g., the cache partitions). Thus, Dean discloses a cache hit rate monitoring apparatus that is used by a cache controller to reallocate the cache partitions: each cache controller allowed to participate in cache-partition reallocation would be provided with the cache hit rate, so that it could perform the partitioning. The mechanism is illustrated in Figure 1, which is shown below with its associated description:



Tag 150 comprises a tag memory 600, a tag allocation register 151, and *N* hit/miss indications 190. **The *N* hit/miss indications 190 are coupled to the cache 130 and serve to inform the cache or process of a hit or miss.** **Tag allocation register 151 has space reserved to track** system metrics 191, elapsed cycles 192, a cache evaluation period 193, **a cache miss percentage 194**, and a cache reallocation factor 195. Additionally, other information may be

tracked and variables added to best allocate the cache ways on a per process basis. System metrics 191 in this example and most preferably would contain M variables, one for each process. Each variable would contain the number of misses for each process. For example, if process 110 missed 100 times, then “100” would be stored in one of the variables in system metrics 191. Alternatively, other system metrics, such as overall system performance, may be stored in system metrics 191. Elapsed cycles 192 is used to determine the number of cycles that have elapsed since the last evaluation of system performance. Cache evaluation period 193 determines when the system's performance should be re-evaluated. If the elapsed cycles 192 is larger than (or equal to) cache evaluation period 193, then the system performance should be evaluated and the ways of the cache reassigned to the processes.

Cache miss percentage 194 is a factor used to determine if the cache should be reallocated. The total misses for all of the process are compared to the cache miss percentage to determine if there are too many misses. The cache miss percentage 194 thus reduces the number of times the cache is reallocated. The cache miss percentage 194 is optional, but preferred. The cache reallocation factor 195 is a factor used to reallocate the cache, and it determines what percentage of the cache will be reallocated to a task. This factor will generally be a certain number of ways of the cache or a certain percentage of the cache.

[...]

Tag allocation controller 161 periodically evaluates system 100's performance and then reallocates ways of the cache to the processes that would be best suited to have more cache memory. Most preferably, this is performed through reference

to the number of misses per process, with the process having the highest number of misses receiving the most allocation of the cache. Alternatively, other system metrics may be used to determine cache allocation. In particular, the operating system of system 100 could determine an overall system performance metric and use this metric to adjust the cache allocation. For example, if the system performance is lowered by assigning more cache to one of the processes, the operating system can determine this fact and assign less cache allocation to this particular process. Any method for determining the allocation of the cache may be used with the current invention.

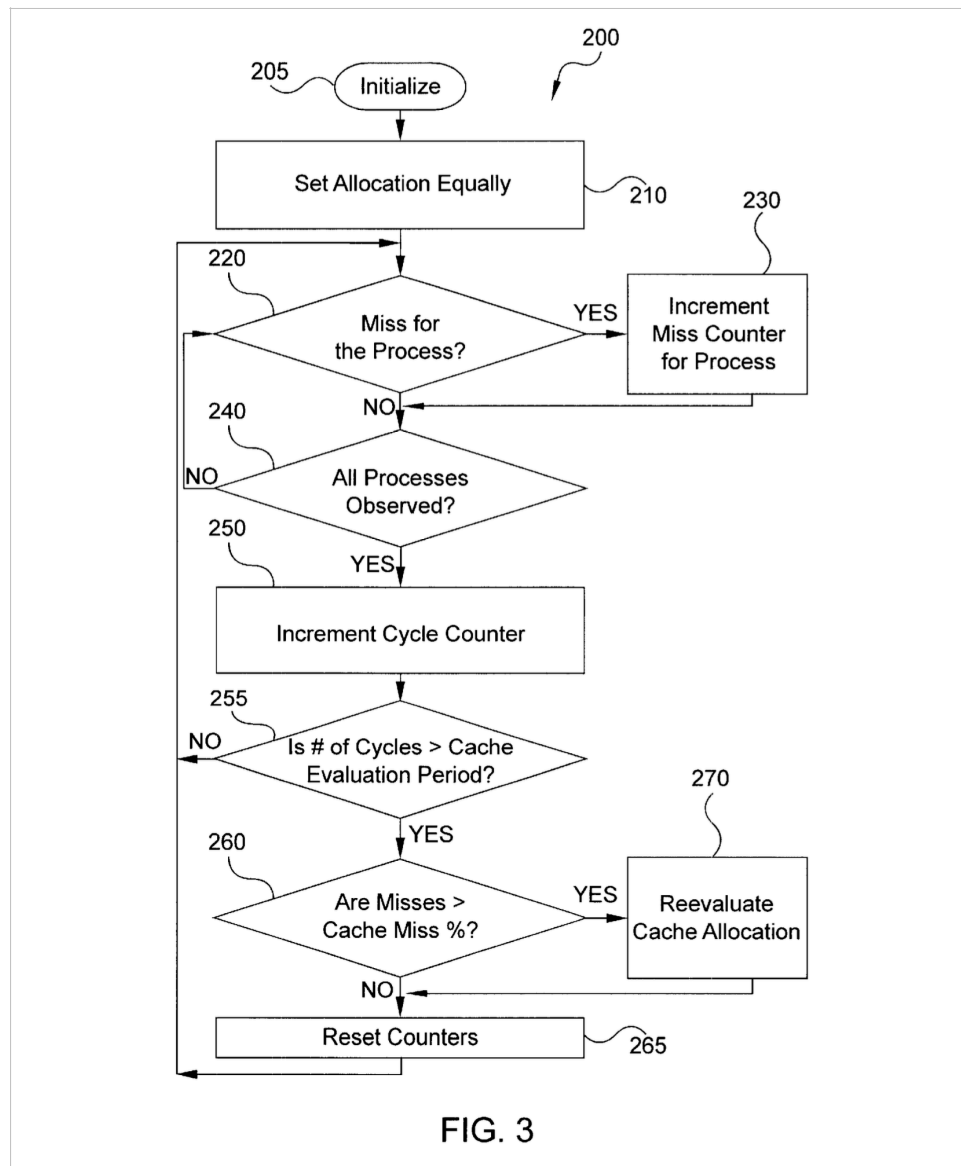
The most preferred method for determining cache allocation is by determining the process that has the most cache misses over a predetermined time period. This method is preferred because it is simple to implement and will usually provide a high amount of system performance. In general, assigning the largest amount of cache to the process with the most cache misses will result in lower overall misses and an increase in system performance. Thus, the cache miss metric will be the metric discussed herein that is used to allocate the ways of the cache, although other metrics may be used. [Dean, 4:24–7:16, emphasis added]

(209) Dean's description above confirms the understanding that the preceding figure shows a partitionable cache, and it confirms the understanding that Dean's system uses a cache-miss percentage to allocate storage in the cache to different processes or processors. Each cache controller allowed to participate in cache-

partition reallocation would be provided with the cache hit rate, so that it could perform the partitioning. Moreover, a cache miss rate is simply the inverse of the cache hit rate ($miss_rate = 1 - hit_rate$).

(210) As the figure above discloses, Dean's cache-miss counters are held in close proximity with the cache's tag arrays. For instance, the cache-miss percentage 194 is held in Tag 150. As described previously, in the combination of Cherabuddi and Dean, the cache-miss hardware, just as in Dean, would be held with the cache's tag arrays, e.g., in close proximity with Cherabuddi's tag arrays 61, 62, 63, and 64. Therefore, each of Cherabuddi's cache controllers 43a and 43b would have access to and would be able to use the cache-miss hardware; each cache controller is provided with cache hit rate monitoring apparatus.

(211) Dean further describes the use of the cache-miss ratio:



Turning now to FIGS. 3 and 4 in addition to FIG. 1, FIGS. 3 and 4 outline a method for dynamically allocating ways of a multiport cache in accordance with a preferred embodiment of the present invention. Method 200 of FIG. 3 is used by a tag allocation controller 161 to periodically allocate the cache amongst a number of processes, and based on a system metric 191 of performance. In method 200, the system metric 191 is preferably M variables, one for each process, that

track the number of cache misses for each process. In the examples of FIGS. 8 through 10, it will be assumed that there are 4 processes, although the methods of FIGS. 3 and 4 will discuss two processes for simplicity. Basically, ***the methods of FIGS. 3 and 4 keeps track of the system metric 191 while waiting for a certain time. After this waiting period, the system 100 will determine, for the most preferred system metric, the cache miss ratios for each of the processes 110 through 112. Using these ratios, the tag allocation controller 161 performs the methods of FIGS. 3 and 4 to reallocate the cache.*** The system reallocates the cache based on the system metric.

[...]

When the cache evaluation period has completed (step 255=YES), ***it is first determined if the total number of cache misses is large enough to warrant a cache reallocation cycle (step 260). This is done by adding miss counters A, B, C, and D in system metric 191 and comparing the result to a cache miss percentage 195. The cache miss percentage 195 can be calculated by multiplying the expected or desired cache hit ratio for each process by the cache evaluation period and adding up the results. If the total number of misses is less than the cache miss percentage, this indicates that the current cache allocation is working and the current cache allocation should be left alone.*** In this event (step 260=NO), the counters are reset for the current cache cycle counter 192 and the system metric counter(s) 191 (step 265), and another cache evaluation period is started (step 220).

If the sum of the cache miss counters in system metric 191 is greater than the cache miss percentage (step 260=YES), step

270 is performed. Method 270 is shown in FIG. 4, and begins when it is decided if the difference between process cache miss counters is enough to warrant a reallocation cycle (step 271).

[Dean, 8:14–9:48, emphasis added]

(212) Dean’s description above further confirms the understanding that the preceding figure shows a partitionable cache, and it confirms the understanding that Dean’s system uses a cache-miss percentage to allocate storage in the cache to different processes or processors. Each cache controller allowed to participate in cache-partition reallocation would be provided with the cache hit rate, so that it could perform the partitioning. Moreover, a cache miss rate is simply the inverse of the cache hit rate ($miss_rate = 1 - hit_rate$).

(213) It is my opinion that the combination of Cherabuddi and Dean discloses all the elements of Claim 4 and thus renders the claim obvious.

2. Claim 5

(214) Claim 5 reads as follows:

5. A method of dynamically allocating cache on a multiple-processor integrated circuit, where the multiple processor integrated circuit is used in a partitionable multiple-processor system and comprises:

a plurality of processors each coupled to receive instructions from a first level cache associated therewith,

a plurality of allocable upper level cache memory blocks, interconnect apparatus for transmitting cache misses at each first level cache to upper level cache memory blocks assigned thereto, and

allocation apparatus for assigning upper level cache memory blocks to processors;

the method comprising the steps of:

monitoring past cache performance associated with processors and partitions;

determining desired processor to partition and upper level cache block allocations to processors; and

repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits,

wherein the upper level cache blocks are usable to store data and instructions fetched from a random-access main memory.

(215) The following sections analyze this claim in light of the prior art.

- a) A method of dynamically allocating cache on a multiple-processor integrated circuit, where the multiple processor integrated circuit is used in a partitionable multiple-processor system and comprises:*
- b) a plurality of processors each coupled to receive instructions from a first level cache associated therewith,*
- c) a plurality of allocable upper level cache memory blocks,*
- d) interconnect apparatus for transmitting cache misses at each first level cache to upper level cache memory blocks assigned thereto, and*
- e) allocation apparatus for assigning upper level cache memory blocks to processors;*

(216) As shown above, these limitations are all disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference. Therefore, the combination of Cherabuddi and Dean also discloses these limitations.

- f) the method comprising the steps of: monitoring past cache performance associated with processors and partitions; determining desired processor to partition and upper level cache block allocations to processors; and repartitioning the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits,*

(217) As shown above, these limitations are disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference.

(218) The combination of Cherabuddi and Dean also discloses these limitations.

As described previously, Dean's system uses a cache-miss percentage to monitor past cache performance associated with processors and partitions, and uses this information to allocate storage in the cache to the different processors. The system is repartitioned periodically, as performance measurements warrant. Dean describes the use of the cache-miss percentage below:

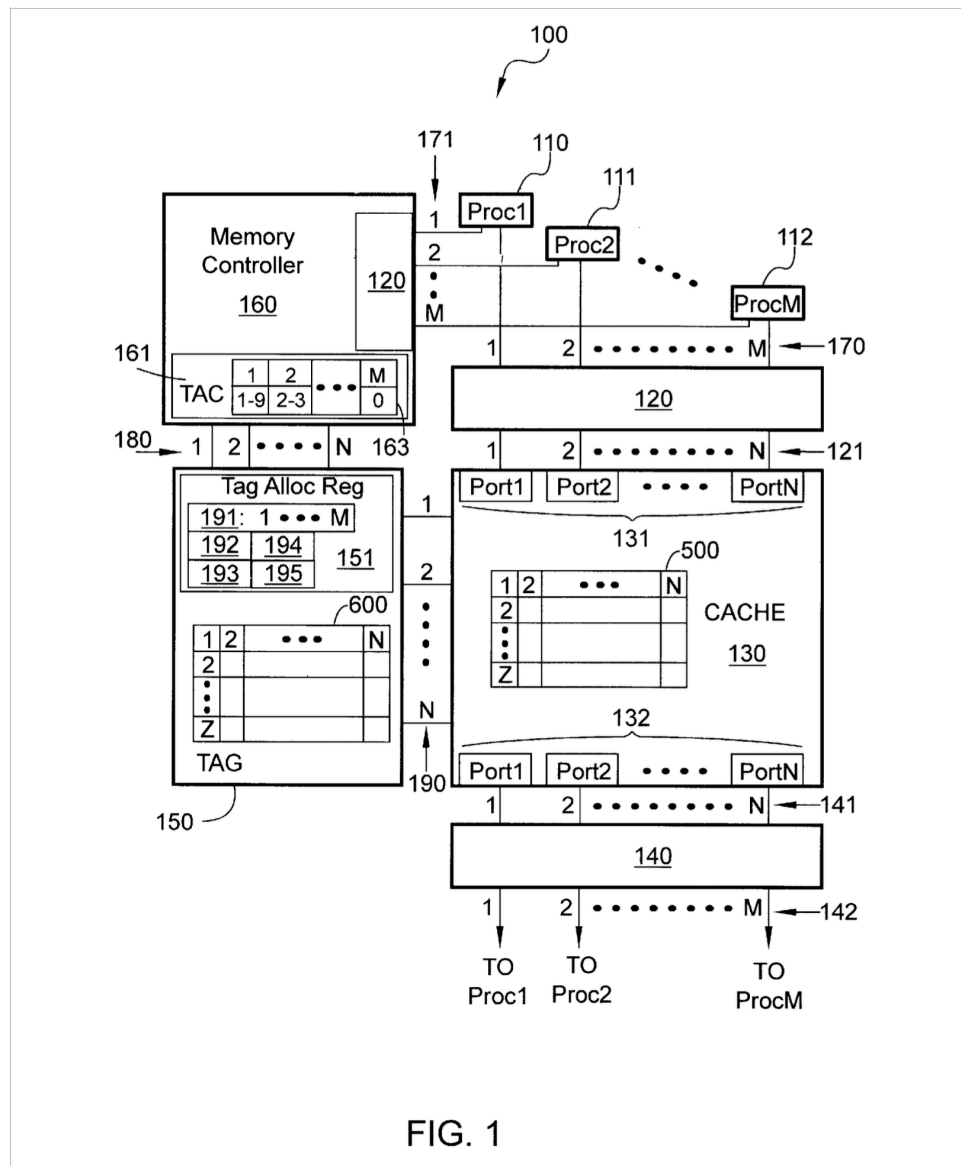
The present invention provides a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and multiple single-cycle cache accesses from a multiport cache. ***Therefore, multiple processes, which could be processors, tasks, or threads can access the cache during any cycle. Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways***, but other metrics may also be used. [Dean, *Abstract*, emphasis added]

The preferred embodiments of the present invention provide a performance based system and method for dynamic allocation of a unified multiport cache. A multiport cache system is disclosed that allows multiple single-cycle look ups through a multiport tag and multiple single-cycle cache accesses from a multiport cache. Therefore, multiple processes, which could be processors, tasks, or threads can

access the cache during any cycle. ***Moreover, the ways of the cache can be allocated to the different processes and then dynamically reallocated based on performance. Most preferably, a relational cache miss percentage is used to reallocate the ways, but other system metrics may also be used.*** [Dean, 3:45–57, emphasis added]

(219) Dean's description above confirms the understanding that Dean's system uses a cache-miss percentage to monitor past cache performance associated with processors and partitions, and uses this information to allocate storage in the cache to the different processors (e.g., Dean's system determines desired processor to partition and upper level cache block allocations to processors and repartitions the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits). The system is repartitioned periodically, as performance measurements warrant.

(220) The mechanism is illustrated in Figure 1, which is shown below with its associated description:



Tag 150 comprises a tag memory 600, a tag allocation register 151, and ***N hit/miss indications 190***. ***The N hit/miss indications 190 are coupled to the cache 130 and serve to inform the cache or process of a hit or miss. Tag allocation register 151 has space reserved to track*** system metrics 191, elapsed cycles 192, a cache evaluation period 193, ***a cache miss percentage 194***, and a cache reallocation factor 195. Additionally, other information may be

tracked and variables added to best allocate the cache ways on a per process basis. System metrics 191 in this example and most preferably would contain M variables, one for each process. Each variable would contain the number of misses for each process. For example, if process 110 missed 100 times, then “100” would be stored in one of the variables in system metrics 191. Alternatively, other system metrics, such as overall system performance, may be stored in system metrics 191. Elapsed cycles 192 is used to determine the number of cycles that have elapsed since the last evaluation of system performance. Cache evaluation period 193 determines when the system's performance should be re-evaluated. If the elapsed cycles 192 is larger than (or equal to) cache evaluation period 193, then the system performance should be evaluated and the ways of the cache reassigned to the processes.

Cache miss percentage 194 is a factor used to determine if the cache should be reallocated. The total misses for all of the process are compared to the cache miss percentage to determine if there are too many misses. The cache miss percentage 194 thus reduces the number of times the cache is reallocated. The cache miss percentage 194 is optional, but preferred. The cache reallocation factor 195 is a factor used to reallocate the cache, and it determines what percentage of the cache will be reallocated to a task. This factor will generally be a certain number of ways of the cache or a certain percentage of the cache.

[...]

Tag allocation controller 161 periodically evaluates system 100's performance and then reallocates ways of the cache to the processes that would be best suited to have more cache memory. Most preferably, this is performed through reference

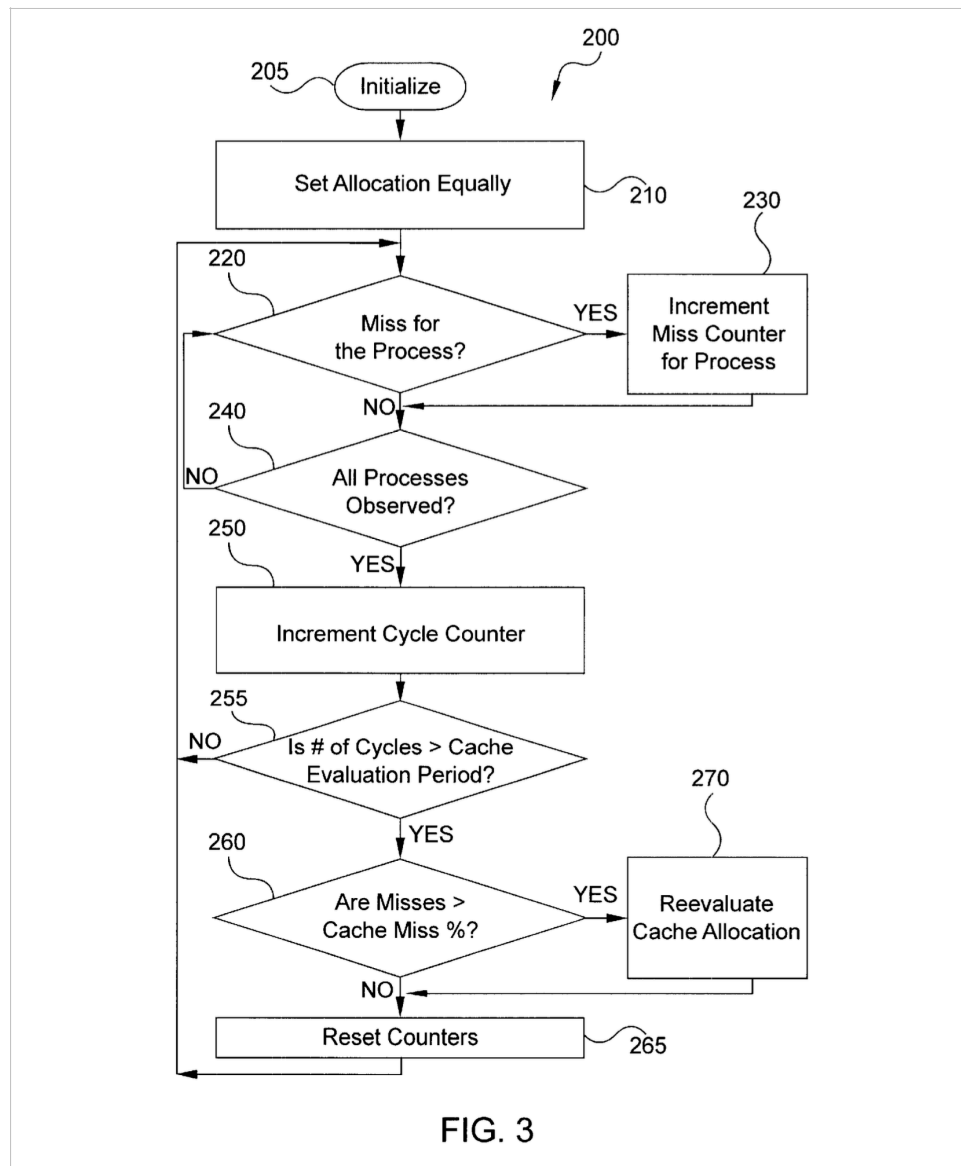
to the number of misses per process, with the process having the highest number of misses receiving the most allocation of the cache. Alternatively, other system metrics may be used to determine cache allocation. In particular, the operating system of system 100 could determine an overall system performance metric and use this metric to adjust the cache allocation. For example, if the system performance is lowered by assigning more cache to one of the processes, the operating system can determine this fact and assign less cache allocation to this particular process. Any method for determining the allocation of the cache may be used with the current invention.

The most preferred method for determining cache allocation is by determining the process that has the most cache misses over a predetermined time period. This method is preferred because it is simple to implement and will usually provide a high amount of system performance. In general, assigning the largest amount of cache to the process with the most cache misses will result in lower overall misses and an increase in system performance. Thus, the cache miss metric will be the metric discussed herein that is used to allocate the ways of the cache, although other metrics may be used. [Dean, 4:24–7:16, emphasis added]

(221) Dean's description above confirms the understanding that Dean's system uses a cache-miss percentage to monitor past cache performance associated with processors and partitions, and uses this information to allocate storage in the cache to the different processors (e.g., Dean's system determines desired processor to

partition and upper level cache block allocations to processors and repartitions the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits). The system is repartitioned periodically, as performance measurements warrant.

(222) Dean further describes the use of the cache-miss ratio:



Turning now to FIGS. 3 and 4 in addition to FIG. 1, FIGS. 3 and 4 outline a method for dynamically allocating ways of a multiport cache in accordance with a preferred embodiment of the present invention. Method 200 of FIG. 3 is used by a tag allocation controller 161 to periodically allocate the cache amongst a number of processes, and based on a system metric 191 of performance. In method 200, the system metric 191 is preferably M variables, one for each process, that

track the number of cache misses for each process. In the examples of FIGS. 8 through 10, it will be assumed that there are 4 processes, although the methods of FIGS. 3 and 4 will discuss two processes for simplicity. Basically, ***the methods of FIGS. 3 and 4 keeps track of the system metric 191 while waiting for a certain time. After this waiting period, the system 100 will determine, for the most preferred system metric, the cache miss ratios for each of the processes 110 through 112. Using these ratios, the tag allocation controller 161 performs the methods of FIGS. 3 and 4 to reallocate the cache.*** The system reallocates the cache based on the system metric.

[...]

When the cache evaluation period has completed (step 255=YES), ***it is first determined if the total number of cache misses is large enough to warrant a cache reallocation cycle (step 260). This is done by adding miss counters A, B, C, and D in system metric 191 and comparing the result to a cache miss percentage 195. The cache miss percentage 195 can be calculated by multiplying the expected or desired cache hit ratio for each process by the cache evaluation period and adding up the results. If the total number of misses is less than the cache miss percentage, this indicates that the current cache allocation is working and the current cache allocation should be left alone.*** In this event (step 260=NO), the counters are reset for the current cache cycle counter 192 and the system metric counter(s) 191 (step 265), and another cache evaluation period is started (step 220).

If the sum of the cache miss counters in system metric 191 is greater than the cache miss percentage (step 260=YES), step

270 is performed. Method 270 is shown in FIG. 4, and begins when it is decided if the difference between process cache miss counters is enough to warrant a reallocation cycle (step 271).

[Dean, 8:14–9:48, emphasis added]

(223) Dean's description above further confirms the understanding that Dean's system uses a cache-miss percentage to monitor past cache performance associated with processors and partitions, and uses this information to allocate storage in the cache to the different processors (e.g., Dean's system determines desired processor to partition and upper level cache block allocations to processors and repartitions the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits). The system is repartitioned periodically, as performance measurements warrant.

(224) Dean describes below how the system is repartitioned, through reallocation of cache blocks to processors:

If, for example, there are a lot of cache misses, uniformly distributed among all the processes, then the current allocation is considered fair and no reallocation is done. Preferably, the cache is only reallocated when there is a relatively significant disparity in cache miss ratios between processes. Because of this, each cache miss counter for each processor in system

metric 191 is compared to the others (step 22). If, for example, process A's miss counter is larger than processor B's miss counter by a pre-determined cache reallocation factor 195, some ways of the cache will be assigned to processor A (step 273). Preferably, this is done by determining which process has the largest percentage of the cache and taking cache ways from this process (step 273).

Alternatively, the method could determine which process had the least misses during the current cache evaluation cycle and then take cache ways assigned to this process. For the latter method, there may be a minimum number of cache ways that will also factor into this analysis: if the method determines that the process with the lowest number of cache hits currently is assigned the minimum number of ways, the method will leave the cache allocation for this process alone.

The pre-determined cache reallocation factor is a number that indicates how much greater the misses for one process must be versus another process for the method to allocate cache ways to the first process. This factor is preferable to prevent the method from continually reassigning the ways of the cache for small differences in miss rates between the processes.

In step 273, it is possible that allocation of the cache ways may be made between many processes. In general, however, for each cache evaluation period, it is recommended that only one process be assigned additional cache ways (and one process less cache ways). This allows the next cache evaluation period to be used to determine the efficacy of the previous assignment.

This will hopefully cause the performance of process A (in the example where a portion of process B's cache allocation is given to process A) to improve without substantially degrading

process B's performance. If, after following cache evaluation periods, process A miss ratios are still significantly higher than process B's, the reallocation will continue until the miss ratio disparity is reduced. Once the reallocation is completed (step 174), all the processor counters and the reallocation period counters are reset (step 265) so that a new cache evaluation period may begin (step 220). [Dean, 9:49–10:29, emphasis added]

(225) Dean's description above further confirms the understanding that Dean's system uses a cache-miss percentage to monitor past cache performance associated with processors and partitions, and uses this information to allocate storage in the cache to the different processors (e.g., Dean's system determines desired processor to partition and upper level cache block allocations to processors and repartitions the system, the step of repartitioning the system including allocation of upper level cache blocks to processors of at least one of the multiple processor integrated circuits). The system is repartitioned periodically, as performance measurements warrant.

g) wherein the upper level cache blocks are usable to store data and instructions fetched from a random-access main memory.

(226) As shown above, this limitation is disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference. Therefore, the combination of Cherabuddi and Dean also discloses this limitation.

(227) It is my opinion that the combination of Cherabuddi and Dean discloses all the elements of Claim 5 and thus renders the claim obvious.

3. Claim 8

(228) Claim 8 reads as follows:

8. The method of claim 5, wherein the multiple processor integrated circuit further comprises a plurality of non-allocable cache memory blocks.

(229) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

(230) As shown above, the limitations for Claim 5 are rendered obvious by Cherabuddi, and rendered obvious by Cherabuddi and Dean, the analysis and evidence for which is incorporated here by reference.

b) wherein the multiple processor integrated circuit further comprises a plurality of non-allocable cache memory blocks.

(231) As shown above, this limitation is disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference.

(232) The combination of Cherabuddi and Dean also discloses this limitation. Dean describes a system in which selected cache blocks (“ways”) are assigned to

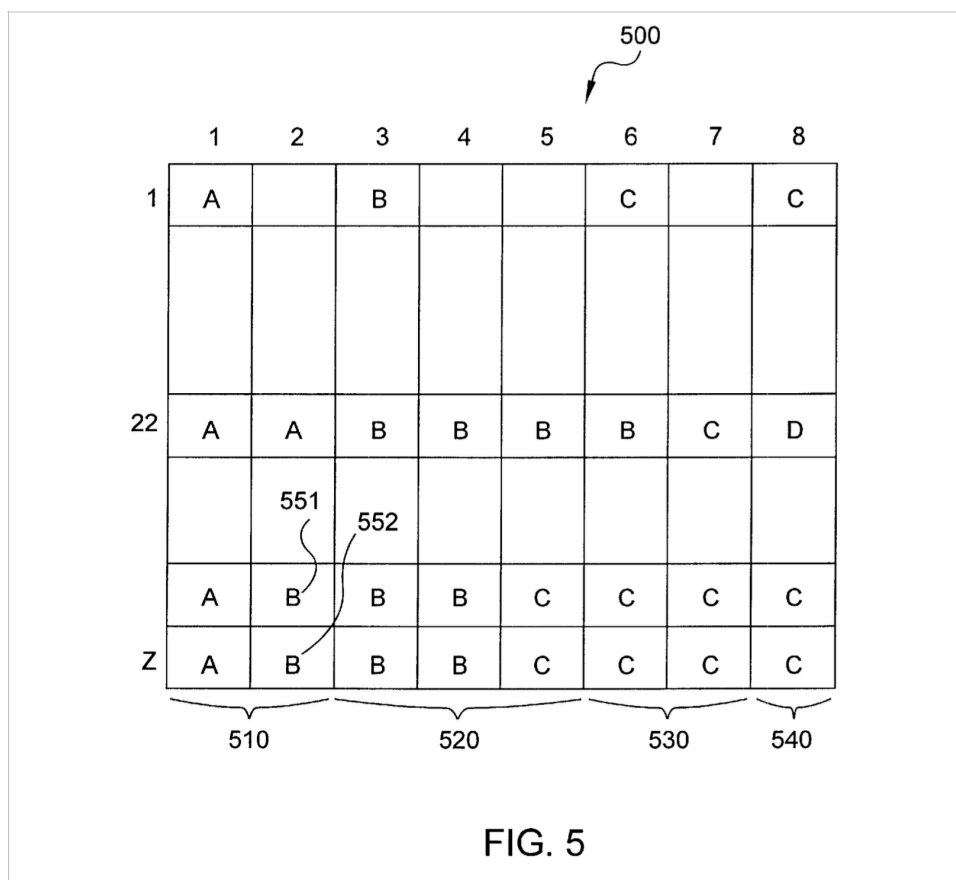
specific processes and when so assigned are unallocable to others. Dean provides a high-level overview below:

The system and method preferably determines which process could benefit the most if additional cache space was allocated to the process or could benefit a computer system the most if additional cache space was allocated to the process. The system and method then allocates additional ways to the process, at the expense of removing ways from other processes. The reallocation can take place periodically, thereby maximizing the performance of the system and handling unforeseen demands by processes. Because the multiport tag can look up multiple requests during one cycle, there is reduced waiting for cache access and look up by processes. Moreover, because of the multiported nature of this system and method, multiple accesses by processes are allowed during single cycles. This reduces cache access time and improves overall system performance. [Dean, 3:59–4:5, emphasis added]

(233) Dean shows that the cache blocks are assigned to different processes, and the portions of the cache assigned to one process may not be allocated to another. For instance, Dean says explicitly that “the system and method preferably determines which process could benefit the most if additional cache space was allocated to the process or could benefit a computer system the most if additional cache space was allocated to the process. The system and method then allocates additional ways to

the process, ***at the expense of removing ways from other processes.***” Thus,

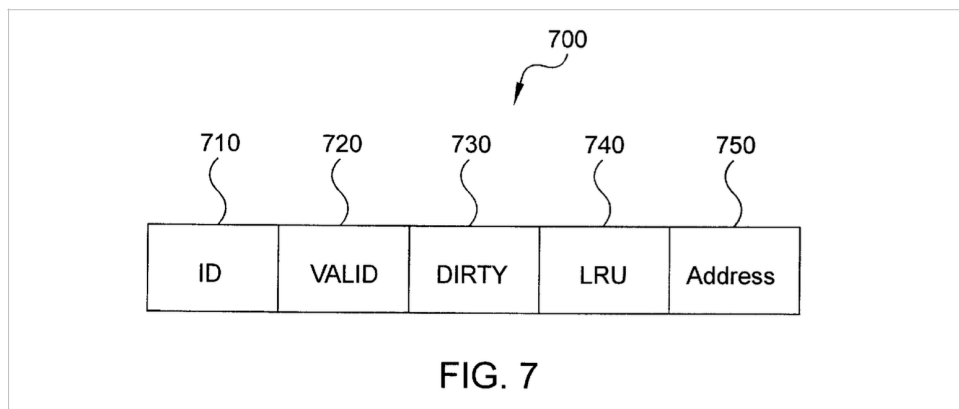
Dean discloses a plurality of non-allocable cache memory blocks.



Referring now to FIG. 5 in addition to FIG. 1, FIG. 5 shows an exemplary cache memory 500. Cache memory 500 is an eight-way memory. There are Z cache lines, and each cache line has eight ways. ***The ways are currently divided such that process A is assigned two ways 510, process B is assigned three ways 520, process C is assigned two ways 520, and process D is assigned one way 540. In this exemplary system, there are four processes 110 through 112, which are processors.***

In this system, a process may be allocated a particular

proportion of the cache. However, until the process writes to the cache, data from other processes may reside in the ways assigned to this process. For example, process A is assigned ways 510. Process B currently has information in locations 551 and 552. Until process A writes over these data, process B will be able to read this data. However, process B will not be able to write data to these locations. [Dean, 7:17–34, emphasis added]



An exemplary tag entry 700 is shown in FIG. 7. For each way in cache memory 500, there will preferably be one entry 700. Entry 700 comprises a process identification (ID) 710, a valid indication 720, a dirty indication 730, a Least Recently Used (LRU) indication 740, and a tag address 750. *The ID 710 contains data that indicates which of the M processes is assigned to this particular way and this line. Once a particular process is assigned a way of a line, no other process can write to that way of that line. Another process can read from the way of the line, if the line contains data for this other process.* Moreover, the process ID 710 allows the tag 150 to determine if a particular way of a line in the cache memory 500 belongs to a process. The valid indication 720 is generally a bit, and is used to determine whether the way of the line is

valid. The dirty indication 730 is also usually a bit, and it is used to mark the way of the line as “dirty”, meaning that the cache contains a more recent version of the data. This version should be written back to main memory in accordance with cache coherence protocols.

[Dean, 7:48–67, emphasis added]

(234) The discussions and figures above confirm the understanding that the cache blocks are assigned to different processes, and that the portions of the cache assigned to one process may not be allocated to another. For instance, Dean discloses in the passages above that “process B ***will not be able to write data to these locations***” and “Once a particular process is assigned a way of a line, ***no other process can write to that way of that line.***” Thus, Dean discloses a plurality of non-allocable cache memory blocks.

(235) It is my opinion that the combination of Cherabuddi and Dean discloses all the elements of Claim 8 and thus renders the claim obvious.

4. Claim 9

(236) Claim 9 reads as follows:

9. The method of claim 5, wherein the interconnect apparatus further comprises a plurality of upper level cache controllers, and where each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache.

(237) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

(238) As shown above, the limitations for Claim 5 are rendered obvious by Cherabuddi, and rendered obvious by Cherabuddi and Dean, the analysis and evidence for which is incorporated here by reference.

b) wherein the interconnect apparatus further comprises a plurality of upper level cache controllers,

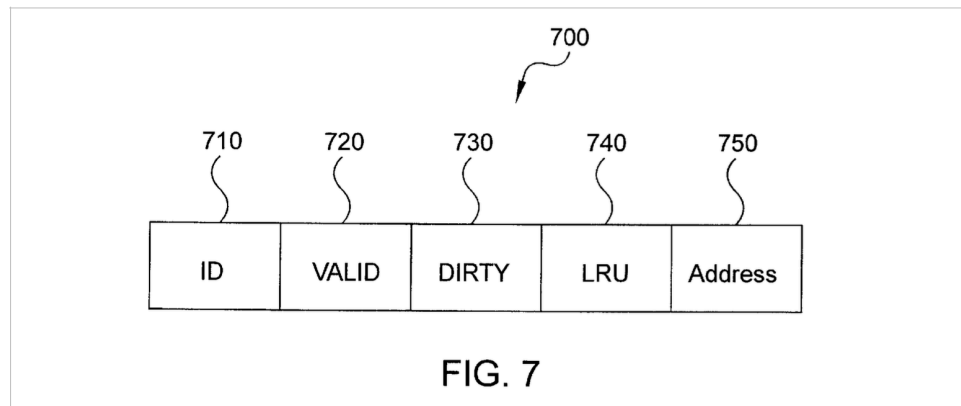
c) and where each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache.

(239) As shown above, these limitations are disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference. Therefore, the combination of Cherabuddi and Dean also discloses these limitations.

(240) The combination of Cherabuddi and Dean also discloses that each upper level cache controller is capable of controlling operation of the allocable cache memory blocks as a writeback cache, by disclosing that its cache is indeed writeback:

In a write-back cache, the entries in the cache directory are comprised of a tag and a number of tag state bits for each of the lines in each set. As before, the tag comprises the upper address bits of the

particular page in main memory from which the copy originated. The tag state bits determine the status of the data for a respective line, i.e., whether the data is invalid, exclusively owned and modified, exclusively owned and unmodified, or shared. [Dean, 2:32–39, emphasis added]



An exemplary tag entry 700 is shown in FIG. 7. For each way in cache memory 500, there will preferably be one entry 700. Entry 700 comprises a process identification (ID) 710, a valid indication 720, **a dirty indication 730**, a Least Recently Used (LRU) indication 740, and a tag address 750. The ID 710 contains data that indicates which of the M processes is assigned to this particular way and this line. Once a particular process is assigned a way of a line, no other process can write to that way of that line. Another process can read from the way of the line, if the line contains data for this other process. Moreover, the process ID 710 allows the tag 150 to determine if a particular way of a line in the cache memory 500 belongs to a process. The valid indication 720 is generally a bit, and is used to determine whether the way of the line is valid. **The dirty indication 730 is also usually a bit, and it is used to mark the way of the line as “dirty”, meaning that the cache contains a more recent version**

of the data. This version should be written back to main memory in accordance with cache coherence protocols. [Dean, 7:48–67, emphasis added]

(241) Dean’s description above confirms the understanding that the caches are writeback. For instance, Dean explicitly discloses “a write-back cache” and states that the tag bits in the cache include “a dirty indication 730 ... meaning that the cache contains a more recent version of the data. This version should be **written back** to main memory in accordance with cache coherence protocols.” This indicates that Dean’s cache is a writeback cache.

(242) It is my opinion that the combination of Cherabuddi and Dean discloses all the elements of Claim 9 and thus renders the claim obvious.

5. Claim 10

(243) Claim 10 reads as follows:

10. The method of claim 5, wherein each upper level allocable cache block further comprises tag memory and cache data memory.

(244) The following sections analyze this claim in light of the prior art.

a) The method of claim 5,

(245) As shown above, the limitations for Claim 5 are rendered obvious by Cherabuddi, and rendered obvious by Cherabuddi and Dean, the analysis and evidence for which is incorporated here by reference.

b) wherein each upper level allocable cache block further comprises tag memory and cache data memory.

(246) As shown above, this limitation is disclosed by Cherabuddi, the analysis and evidence for which is incorporated here by reference.

(247) The combination of Cherabuddi and Dean also discloses this limitation in Figure 1, shown below; in the figure, the tag memory blocks are labeled as elements 600 in TAG 150; the cache data blocks are labeled as elements 500 in CACHE 130. Dean's description of the figure is included as well.

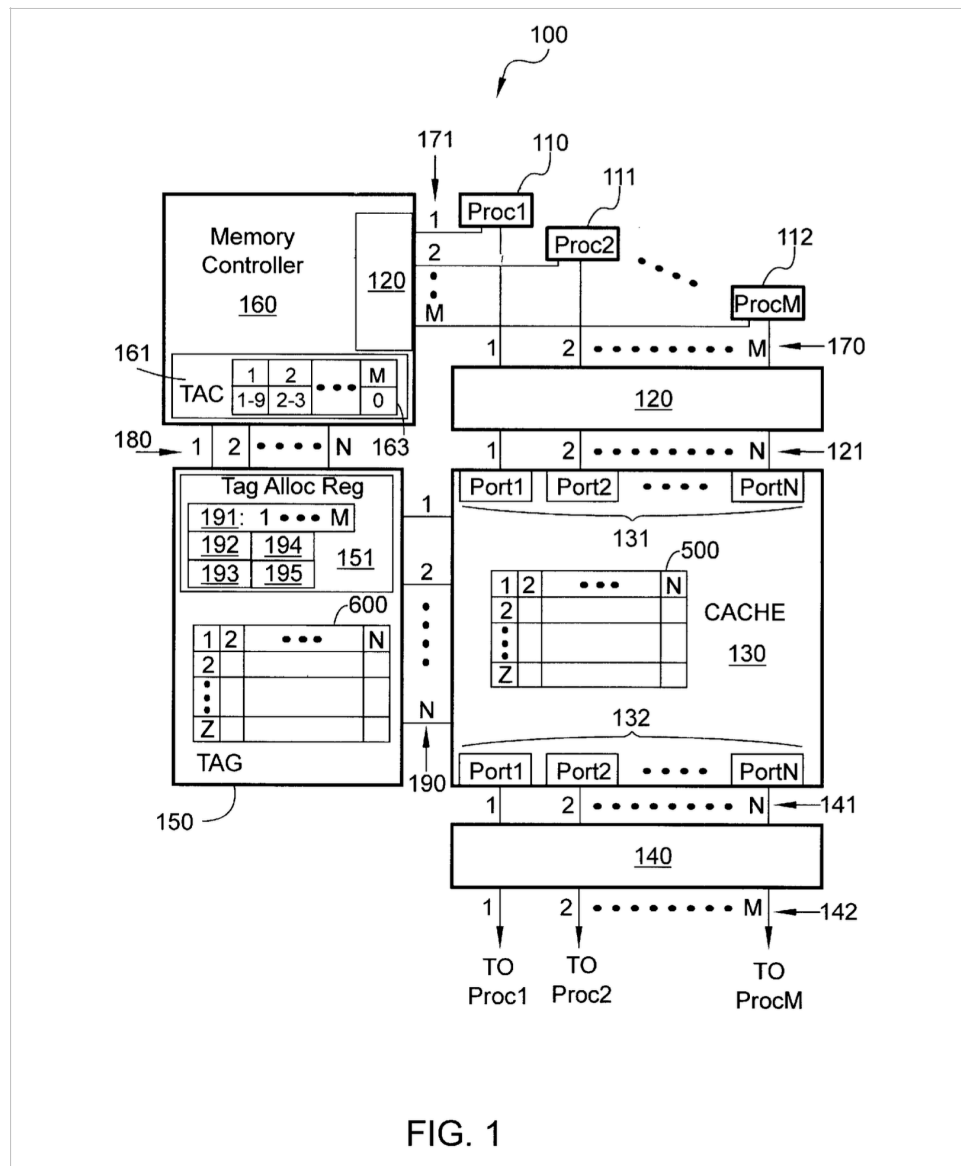


FIG. 1

Tag 150 comprises a tag memory 600, a tag allocation register 151, and N hit/miss indications 190. The N hit/miss indications 190 are coupled to the cache 130 and serve to inform the cache or process of a hit or miss. Tag allocation register 151 has space reserved to track system metrics 191, elapsed cycles 192, a cache evaluation period 193, a cache miss percentage 194, and a cache reallocation factor 195. Additionally, other information may be tracked and variables added to

best allocate the cache ways on a per process basis. System metrics 191 in this example and most preferably would contain M variables, one for each process. Each variable would contain the number of misses for each process. For example, if process 110 missed 100 times, then “100” would be stored in one of the variables in system metrics 191. Alternatively, other system metrics, such as overall system performance, may be stored in system metrics 191. Elapsed cycles 192 is used to determine the number of cycles that have elapsed since the last evaluation of system performance. Cache evaluation period 193 determines when the system's performance should be re-evaluated. If the elapsed cycles 192 is larger than (or equal to) cache evaluation period 193, then the system performance should be evaluated and the ways of the cache reassigned to the processes.

[...]

Memory controller 160 comprises a tag allocation controller mechanism 161 that allocates the cache based on system performance. Tag allocation controller 161 comprises a way assignment memory 163. Way assignment memory 163 contains data that indicates which process is assigned which way. Most preferably, cache memory 500 is assigned on a way-by-way basis and not a line-by-line basis. In other words, each of the processes will be assigned to zero, one, or more ways and will be allowed to write to its way. All lines in a way would be assigned to one or more processes. In the example of FIG. 1, way assignment memory 163 comprises M entries (indicated by 1, 2, . . . , M in the figure). Each of these entries contains the way mapping for each process. For instance, process 1 is mapped to ways 1 and 9, process 2 is mapped to ways 2 and 3, and process M is mapped to zero ways. Preferably, there are less processes than there are ways and each process is assigned at least one way. However, ***if***

there are processes that need a large amount of cache space, the tag allocation controller 161 can increase the number of ways allocated to these processes. There may be times when another, rarely-used or idle process would not have any ways allocated to it. This is the situation shown in FIG. 1. In general, each process will have at least one way assigned to it.

[...]

Memory controller 160 communicates the address (or a portion of the address) to the tag 150 through bus 180. Each of the N ports of bus 180 also preferably contains data that indicates which process is accessing a port. For instance, if process 112 is reading from port 3, the fact that process 112 (and not the other M-1 processes) is accessing this port is communicated to the tag 150 through bus 180. ***The tag 150 uses the process information during a read to determine if the data being requested is in the cache. For example, if process 112 is reading from the cache, the tag 150 not only should know the address of the data but should also know which processor is requesting the data, as there could be multiple data per line having the same address but from different processes. The tag 150 uses the process information (called a process identification in reference to FIG. 7) to perform cache look ups.***

Cache 130 comprises N write ports 131, N read ports 132, and cache memory 500. There are N ways in memory 500 and Z lines in the cache. Each of the N write ports 131 can write to any way of the cache. Similarly, each of the N read ports 132 can read from any way of the cache. Processes-to-cache multiplexor 120 essentially arbitrates and multiplexes the M processes to the N ports 121 of the cache. The arbitration occurs because only one process

may have access to one port at one time. The multiplexing occurs because there are M processors and N ports, with M generally larger than N. Processes-to-cache multiplexor 120 is essentially N M-to-N multiplexors that also have arbitration control. Cache-to-processes multiplexor 140 is essentially N N-to-M multiplexors, and operates in reverse of the processes-to-cache multiplexor 120. Each of the multiplexors 120, 140 may be part of cache 130, which is preferred, or may be separate from cache 130, as shown in FIG. 1 but is less preferred. [Dean, 4:24–7:16, emphasis added]

(248) Dean’s description above confirms the understanding that, in the preceding figure, the tag memory blocks are labeled as elements 600 in TAG 150; the cache data blocks are labeled as elements 500 in CACHE 130. For instance, in the passage above Dean states, “Tag 150 comprises a tag memory 600,” and “Memory controller 160 communicates the address (or a portion of the address) to the tag 150 through bus 180,” and “The tag 150 uses the process information during a read to determine if the data being requested is in the cache.”

(249) It is my opinion that the combination of Cherabuddi and Dean discloses all the elements of Claim 10 and thus renders the claim obvious.

D. The Combination of Cherabuddi and Noel

(250) In my opinion, it would have been obvious for a person of ordinary skill to combine Cherabuddi and Noel. The choice would have been obvious, and the

steps to make it happen would have been obvious, as Cherabuddi and Noel are natural complements to one another. Both Cherabuddi and Noel describe a multiprocessor system in which the processors, coupled together, share a level of memory in a way that portions of the memory can be assigned to certain processors and therefore be dedicated to them. While both Cherabuddi and Noel describe the use of operating systems and how they might interact with such a system, Cherabuddi's description seems to indicate that the operating system is unaffected by the partitioning of the system, whereas Noel has explicit descriptions about whether the operating system kernel needs to be rebooted or not.

(251) Cherabuddi and Noel are also both reasonably pertinent to a problem described by the '264 Patent to avoid system crashes or other issues arising in one partition that do not affect the other partitions—"Machines also exist that are capable of simultaneously running several copies of the same operating system with each copy running independently in a separate partition. These machines are advantageous in that each partition may be dedicated to particular users and applications, and problems (including system crashes) that arise in one partition need not adversely affect operation in other partitions." '264 Patent, 2:60-67.

Cherabuddi recognizes the advantages of dynamically allocating cache resources to workaround defective processors. Cherabuddi, 2:40-49 ("The ability to dynamically allocate cache resources between multiple processors advantageously

allows the entire cache memory to be used, irrespective of whether one or both processors are currently active, thereby maximizing cache resources while allowing for both one-thread and two-thread execution modes. In addition, the present invention may be used to maximize cache resources when one of the on-board processors is defective. For example, if one processor is found to be defective during testing, it may be set as inactive, and the cache access circuit may allocate the entire cache memory to the other processor.”). Noel recognizes that in the prior art, a computing system could be partitioned with hardware to make a subset of the resources on a computer available for a particular application, but the resources were not permanently dedicated, which left “issues concerning performance improvements by means of load balancing of resources among partitions and resource availability.” Noel, 2:7-13. Rather than partitioning a computing system and reserving a subset of system resources for a particular application, the invention taught by Noel discusses “multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together” such that the single physical machine, containing multiple physical processors and resources, “is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct, copy, or instance, of an operating system.” Noel, 4:29-36. Noel

recognizes that the “resulting adaptively partitioned, multi-processing (APMP) system exhibits both scalability and high performance.” Noel, 4:54-56.

(252) Cherabuddi describes the benefit of dynamically allocating cache resources between multiple processors, maximizing and efficiently utilizing cache resources. Cherabuddi, 2:40-49, 3:40-46. Noel similarly observed that it “would be advantageous to dynamically allocate cache to processors on a multiple processor integrated circuit, including on such integrated circuits that are parts of partitionable machines, so as to provide an amount of cache appropriate to each thread, or partition, executing on the system.” Noel, 3:8-12. Noel also recognizes the desirability of having “a new computer system which provides improved flexibility, resource availability[,] and scalability.” Noel, 4:23-25. A POSITA would therefore recognize that including dynamically turning on and off CPUs assigned to each partition and changing partitions without rebooting the system as taught by Noel would facilitate the modulation of CPU activity and cache partition allocation, as described by Cherabuddi.

(253) Cherabuddi’s disclosure is consistent with the disclosure of the ’264 Patent that “[i]t would be advantageous to dynamically allocate cache to processors on a multiple processor integrated circuit, including on such integrated circuits that are parts of partitionable machines, so as to provide an amount of cache appropriate

to each thread, or partition, executing on the system.” ’264 Patent, 3:8-12;

Cherabuddi, 1:62-66 (“[T]he resources of a partitioned cache memory are dynamically allocated between two or more processors on an MPU based on a desired system configuration or to the processing needs of the processors.”).

(254) Cherabuddi describes a multiprocessor system as a merging of two processors and a cache whose memory can be divided between the two processors:

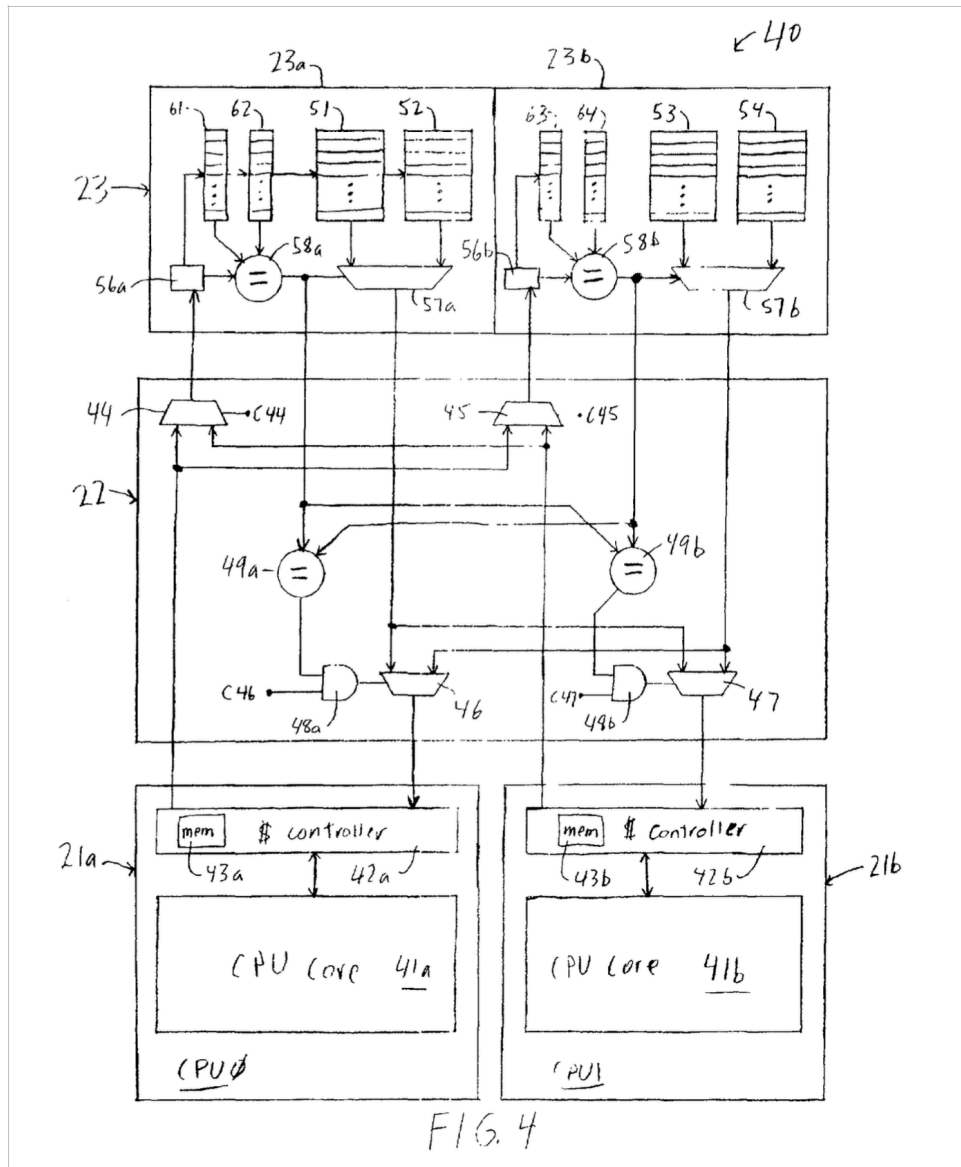
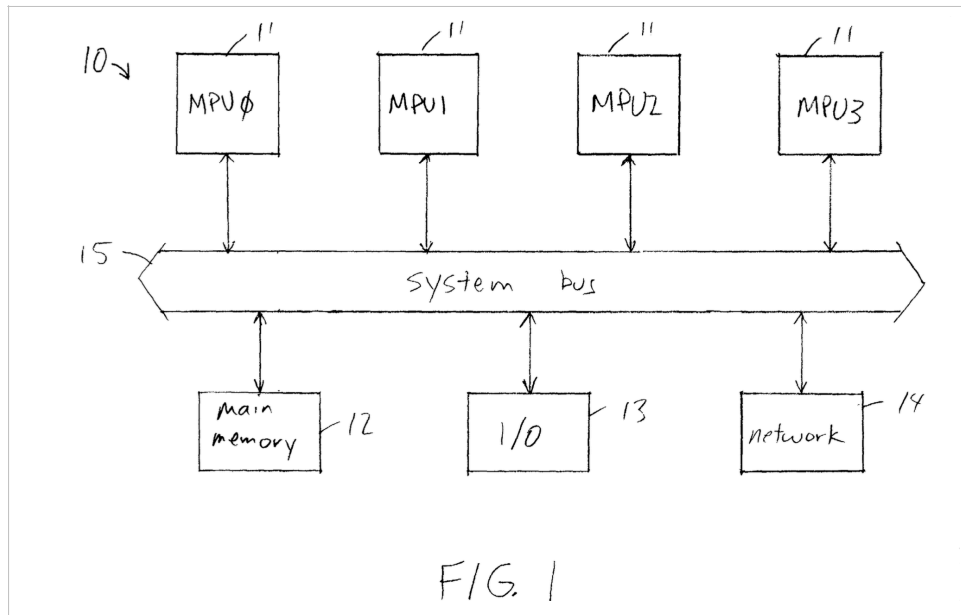


FIG. 4 shows an MPU 40 that is one embodiment of MPU 20, and includes CPUs 21a-21b, cache access circuit 22, and cache memory partitions 23 a-23 b. Each CPU 21 is shown to include a CPU core 41 and a cache controller 42. Each cache controller 42, which may be of conventional architecture, transfers address and data between its associated CPU core 41 and cache access circuit 22, and includes (or is associated with) a memory element 43. Memory element 43 may be

any suitable memory device including, for example, a register or memory cell. Although shown in FIG. 4 as being internal to cache controller 42, memory element 43 may be external to cache controller 42. CPU core 41 includes other well-known elements of CPU 21 including, for instance, L1 cache memory, instruction units, fetch and decode units, execution units, register files, write cache(s), and so on.

Cache memory partition 23 a includes two data RAM arrays 51-52 having corresponding searchable tag arrays 61-62, respectively, while cache memory partition 23 b includes two data RAM arrays 53-54 having corresponding searchable tag arrays 63-64, respectively. Cache memory partition 23 a includes a well-known address converter 56 a that converts a main memory address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 61-62 and the data arrays 51-52. Similarly, cache partition 23 b includes a well-known address converter 56 b that converts an address received from cache access circuit 22 into a cache address that is used to concurrently address the tag arrays 63-64 and the data arrays 53-54. [Cherabuddi, 4:66–5:28]

(255) Cherabuddi says that the purpose of the operating system is to assigns tasks to the two CPU cores:



Referring also to FIG. 1, **system 10 includes well-known system operating software that assigns tasks of one or more computer programs running thereon to the various MPUs 20 for execution. The operating software, which is often referred to as the system kernel, also assigns tasks between the CPUs 21 a-21 b of each MPU 20.** For applications that include a single instruction execution thread and are thus best executed using only one CPU 21, e.g., for applications having a highly sequential instruction code, the kernel assigns all the tasks to one CPU and idles the other CPU. Conversely, for applications that can be divided into two parallel instruction execution threads, e.g., for applications having parallel execution loops, the kernel may assign different threads to CPUs 21 a-21 b for simultaneous execution therein. [Cherabuddi, 3:46–60, emphasis added]

(256) Cherabuddi also describes how the operating system kernel decides how to partition the cache among the processor cores. The mechanism is described by

Cherabuddi as being in either a one-thread (1T) or two-thread (2T) state. In the 1T mode, one thread is running on one CPU core, and that threads gets the entirety of the cache to itself; in the 2T mode, both CPU cores are active, and they shared the cache blocks. The mechanism is described below, with reference to Figure 3 as well as the components of Figures 2 and 4 above

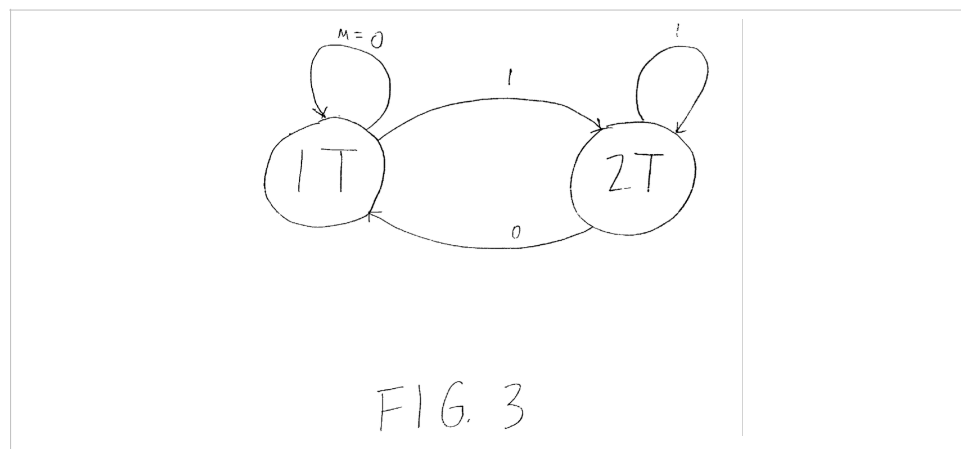


FIG. 3 illustrates state transitions of MPU 20 between a one-thread (1T) state and a two-thread (2T) state. In one embodiment, upon power-up of MPU 20, **the kernel sets a mode signal $M=0$ to initialize MPU 20 to the 1T state. The kernel sets one of the CPUs 21 to an active state and sets the other CPU 21 to an inactive state.** For purposes of discussion herein, during the 1T state, the kernel sets CPU 21 a as the active CPU and sets CPU 21 b as the inactive CPU, although in other embodiments the kernel may set CPU 21 b as the active CPU and set CPU 21 a as the inactive CPU. **While in the 1T state, the kernel assigns tasks of the computer program(s) only to the active CPU 21 a, while the other CPU 21 b remains idle. In response to $M=0$, cache access circuit 22**

couples the first CPU 21 a to both the first and second cache memory partitions 23 a-23 b to allow the first CPU 21 a to use all resources of cache memory 23. In this state, the active CPU 21 a may use cache memory partitions 23 a-23 b as a 4-way associative cache memory.

If, during execution of the computer program(s), the kernel determines that certain tasks may be executed in parallel, and thus may be divided into 2 threads, the kernel may transition MPU 20 to the 2T state by changing the mode signal to M=1. When M=1, the kernel sets both CPUs 21 a-21 b to the active state, and thereafter assigns one execution thread to CPU 21 a and another execution thread to CPU 21 b in a well-known manner. In response to M=1, dirty data in cache memory partition 23 b is written back to main memory 12 using a well-known writeback operation, thereby flushing cache memory partition 23 b. The cache access circuit 22 couples the first CPU 21 a to the first cache memory partition 23 a for exclusive access thereto, and couples the second CPU 21 b to the second cache memory partition 23 b for exclusive access thereto. *In this state, CPU 21 a may use cache memory partition 23 a as a dedicated 2-way associative cache memory, and CPU 21 b may use cache memory partition 23 b as a dedicated 2-way associative cache memory.*

Thereafter, if the kernel determines that only one of CPUs 21 a-21 b is necessary for a particular instruction code sequence, the kernel may transition MPU 20 to the 1T state by changing the mode signal to M=0, flushing the second cache memory partition 23 b, and then assigning execution of the instruction code sequence to the active CPU 21 a. [Cherabuddi, 3:61–4:37, emphasis added]

(257) Cherabuddi does not discuss any scenarios in which the operating system kernel might need to be rebooted. A person of ordinary skill would know that multiprocessor systems have non-trivial coherence issues (see the *Background* section) and might wish to verify for him/herself that the system under construction would or would not require operating system reboots. Therefore, out of an abundance of caution, a person of ordinary skill designing a system based on Cherabuddi would have been motivated to seek out references providing description of a partitionable system, including descriptions of re-partitioning scenarios that require kernel reboots.

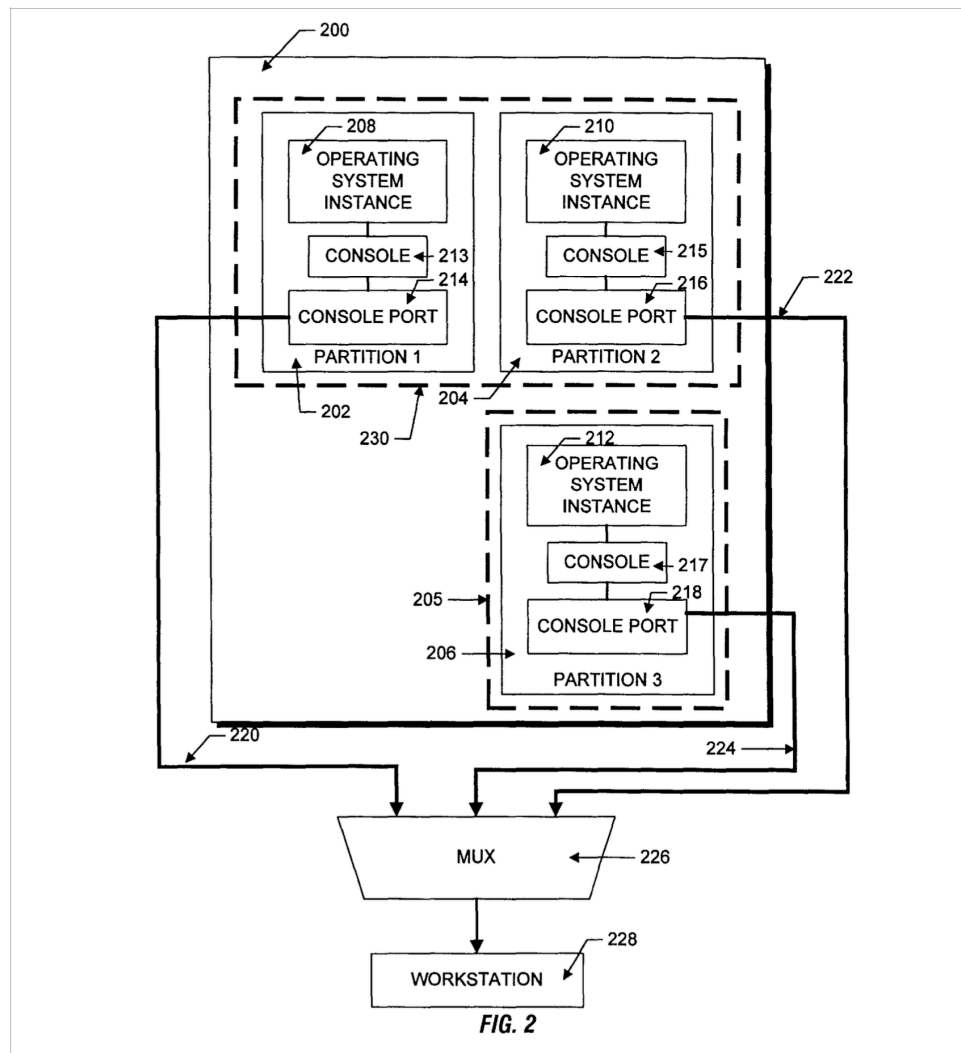
(258) Noel is such a reference: Noel provides a description of a partitionable system, including descriptions of re-partitioning scenarios that require kernel reboots.

(259) Noel's partitionable, multiprocessor computer system is described as follows in the patent's Abstract:

Multiple instances of operating systems execute cooperatively in **a single multiprocessor computer** wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources **is subdivided by software into multiple partitions**, each with the ability to run a distinct copy, or instance, of an operating system. At

different times, different operating system instances may be loaded on a given partition. ***Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration.*** The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. A grouping of partitions, a community, shares memory. ***Memory may be private to a particular partition or may be shared by partitions within a community.*** When a community is formed the creating instance reads a configuration tree and builds management structures for the shared memory owned by the community. A single system may have one or more communities, each with their own representation within the configuration tree. [Noel, *Abstract*, emphasis added]

(260) As Noel says in the passage above, the system's resources, including memory, can be dynamically assigned to different partitions, each of which is running its own instance of an operating system. Noel describes the system in more detail in the following passages:



An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in FIG. 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.

In a preferred embodiment, this allocation is performed by a software program called a “console” program, which, as will hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in FIG. 2 as programs

213, 215 and 217. The console program may be a modification of an existing administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtualize the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in FIG. 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and 212 to use the resources appropriately and provide coordination of resource allocation and sharing. The hardware platform may optionally provide hardware assistance for the division of resources, and may provide fault barriers to minimize the ability of an operating system to corrupt memory, or affect devices controlled by another operating system copy.

The execution environment for a single copy of an operating system, such as copy 208 is called a “partition” 202, and the executing operating system 208 in partition 202 is called “instance” 208. Each operating system instance is capable of booting and running independently of all other operating system instances in the computer system, and can cooperatively take part in sharing resources between operating system instances as described below. [Noel, 6:49–7:19]

(261) As described in the passage above, Noel divides the system into “partitions,” each of which has an instance of an operating system running in it. The

partitioning can be done by the operating system, or the hardware, or a combination of the two.

(262) In the passage below, Noel further explains that, in some instances, the operating system instances need to be rebooted at the time of repartitioning, and in other instances, the operating system instances ***do not*** need to be rebooted. For instance, if the I/O subsystem changes, the various partitions may require reboots, but, if only memory allocations are changed, then it might be done without requiring any rebooting.

Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or ***altering the ownership of memory fragments***. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

However, ***callback functions do not cause any changes to take place on the running operating system instances***. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. ***For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be***

handled by an operating system instance without the need for a reboot. [Noel, 22:8–27, emphasis added]

(263) Cherabuddi describes a system with multiple cores and a cache that can be partitioned among them. Noel describes a partitionable system and provides descriptions of partitioning instances that may and may not need rebooting. It is my opinion that a person of ordinary skill would have found it obvious and straightforward to combine Cherabuddi and Noel. They are in extremely similar fields (they both describe multiprocessor systems); they use similar techniques (they both describe a multiprocessor system in which the memory can be partitioned between the different processors); they are compatible, orthogonal, and easily combined (Noel simply describes in more detail than Cherabuddi the scenarios in which repartitioning the system might require rebooting); and their combination produces nothing more than the expected behavior, which is the sum of both systems. Doing so would have been well within the skill set of the person of ordinary skill.

(264) This modification would have been obvious to a person of ordinary skill for several reasons. First, this modification combines familiar elements (caches, cache partitions, the ability to assign different partitions to different processors, algorithms to guide the assignment, and considerations of operating system involvement) according to known methods that would yield predictable results. Second, this

modification is the addition of one known element (Noel's descriptions of partitioning instances that may and may not need rebooting) into another mechanism (Cherabuddi's partitionable cache that can be partitioned according to a given algorithm) that yields predictable results. Third, it is the use of a known technique (using an algorithm to partition a cache, with the consideration of instances that may and may not need rebooting) to improve similar items or methods (Cherabuddi's partitionable cache) that yields predictable results. Fourth, it is the application of a known technique (a description of scenarios in which repartitioning the system might require rebooting) to a prior art reference (Cherabuddi) that is ready for improvement that yields predictable results. Fifth, the need to partition the cache, already acknowledged in Cherabuddi, is furthered by the use of a cache-partitioning algorithm that includes considerations of operating system involvement, specifically the consideration of instances that may and may not need rebooting, as taught in Noel.

(265) A POSITA would recognize that dynamically turning on and off CPUs assigned to each partition and changing partitions without rebooting the system, as taught by Noel, would facilitate the cache partition allocation, as described by Cherabuddi, because rebooting the system slows down the system and decreases performance. Noel explains that efficient operation of applications in computing environment requires "fast, powerful[,] and flexible computing systems." Noel,

1:15-17. Noel recognizes that this motivation pervades other sought-after efficiency gains, including “expand[ing] the system work capacity and chang[ing] configurations to achieve better utilization of resources without stopping execution of application programs on the system.” Noel, 1:23–27.

E. Claim 11 Is Rendered Obvious by the Combination of Cherabuddi and Noel

(266) In my opinion, the combination of Cherabuddi and Noel renders obvious Claim 11 of the ’264 patent. My analysis, which includes my reasoning and the bases for my opinion, is provided below.

1. Claim 11

(267) Claim 11 reads as follows:

11. The method of claim 10, further comprising the steps of stopping execution of operating systems in each partition, and restarting execution of operating systems in each partition, and wherein the system is capable of being repartitioned without rebooting each operating system.

(268) The following sections analyze this claim in light of the prior art.

a) The method of claim 10,

(269) As shown above, the limitations for Claim 10 are rendered obvious by Cherabuddi, the analysis and evidence for which is incorporated here by reference.

b) further comprising the steps of stopping execution of operating systems in each partition, and restarting execution of operating systems in each partition,

(270) The combination of Cherabuddi and Noel discloses this limitation. As described above, Noel presents a multiprocessor computer system in which the resources, including cache memory space, are partitioned. The system has multiple instances of operating systems executing on it, one copy of an operating system running in each partition:

Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. ***The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system.*** At different times, different operating system instances may be loaded on a given partition. ***Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. A grouping of partitions, a community, shares memory. Memory may be private to a particular partition or may be shared by partitions within a community.*** When a community is formed the creating instance reads a configuration tree and builds management structures for the shared memory owned by the community. A single system may have one or

more communities, each with their own representation within the configuration tree. [Noel, *Abstract*, emphasis added]

In accordance with the principles of the present invention, multiple instances of operating systems execute cooperatively in ***a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.***

[Noel, 4:29–40, emphasis added]

(271) Noel’s description above confirms the understanding of a multiprocessor computer system in which the resources, including cache memory space, are partitioned. The system has multiple instances of operating systems executing on it, one copy of an operating system running in each partition. For instance, in the passages above, Noel states, “The single ***physical machine*** with multiple physical processors and resources ***is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system,***” and “***Resources, such as CPUs and memory, can be dynamically assigned to different partitions*** and used by instances of operating systems

running within the machine by modifying the configuration,” and “In accordance with one embodiment, ***the partitioning of resources is performed by assigning resources within a configuration.***”

(272) Noel discloses that the operating systems in each partition can be stopped and restarted:

In addition, the console program is configured to provide a mechanism to remove a CPU from the available CPUs within a partition in response to a request by an operating system running in that partition. ***Each operating system instance must be able to shutdown, halt, or otherwise crash in a manner that control is passed to the console program. Conversely, each operating system instance must be able to reboot into an operational mode, independently of any other operating system instance.***
[Noel, 9:19–27, emphasis added]

Passive hardware resources which have no owner are unavailable for use until ownership is established. Once ownership is established by altering the configuration tree, the operating system instances may begin using the resources. When an instance makes an initial request, ownership can be changed by causing the owning operating system to stop using a resource or by a console program taking action to stop using a resource in a partition where no operating system instance is executing. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. ***The action required to cause an operating system to stop using a***

hardware resource is operating system specific, and may require a reboot of the operating system instances affected by the change.

[...]

When a resource is de-assigned, the owner may decide to deassign the owner field, or both the owner and current_owner fields. The decision is based on the ability of the owning operating system instance running in the partition to discontinue the use of the resource prior to de-assigning ownership. ***In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current_owner field is not changed. When the owning operating system instance reboots, the console program can clear any current_owner fields for resources that have no owner during initialization.*** [Noel, 18:61–19:38, emphasis added]

(273) Noel’s description above confirms the understanding that the operating systems in each partition can be stopped and restarted. For instance, in the passages above, Noel states, “***Each operating system instance must be able to shutdown, halt, or otherwise crash*** in a manner that control is passed to the console program. Conversely, ***each operating system instance must be able to reboot into an operational mode***, independently of any other operating system instance.” This makes it clear that each operating system instance must be able to shutdown, halt, or otherwise crash, as well as to reboot.

(274) Noel therefore discloses stopping execution of operating systems in each partition, and restarting execution of operating systems in each partition.

c) and wherein the system is capable of being repartitioned without rebooting each operating system.

(275) The combination of Cherabuddi and Noel discloses this limitation. Noel discloses that the system can be repartitioned (e.g., changing the memory allocation of fragments) *without* requiring a reboot and makes clear that even if rebooting is needed, only the affected partitions need to be rebooted (i.e., rebooting each operating system is not required, only the affected partitioned need rebooting):

Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. At different times, different operating system instances may be loaded on a given partition. ***Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration.*** ***The partitions themselves can also be changed without rebooting the system by modifying the configuration tree.*** A grouping of partitions, a community, shares memory. Memory may be private to a particular partition or may be shared by partitions within a community. When a community is formed the creating instance

reads a configuration tree and builds management structures for the shared memory owned by the community. A single system may have one or more communities, each with their own representation within the configuration tree. [Noel, *Abstract*, emphasis added]

Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, ***moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot.*** [Noel, 22:8–27, emphasis added]

(276) Noel's description above confirms the understanding that the system can be repartitioned (e.g., changing the memory allocation of fragments) without requiring a reboot and makes clear that even if rebooting is needed, only the affected

partitions need to be rebooted (i.e., rebooting each operating system is not required, only the affected partitioned need rebooting). For instance, in the passages above Noel states, “**Resources**, such as CPUs and memory, **can be dynamically assigned to different partitions** and used by instances of operating systems running within the machine by modifying the configuration. **The partitions themselves can also be changed without rebooting the system.**” Noel also says in the passages above that “moving an input/output processor from one partition to another **may require both partitions to reboot**. Changing the memory allocation of fragments, on the other hand, **might be handled by an operating system instance without the need for a reboot.**” E.g., in some instances, repartitioning changes to the system might require the rebooting of each operating system involved, and in other instances, the system is capable of being repartitioned without rebooting each operating system.

(277) It is my opinion that the combination of Cherabuddi and Noel discloses all the elements of Claim 11 and thus renders the claim obvious.

Executed on Tuesday, January 14, 2025 in Annapolis, Maryland



Dr. Bruce Jacob

REFERENCES

L. Hammond, B. Nayfeh, and K. Olukotun, 1997. “A single-chip multiprocessor,” *Computer*, vol. 30, no. 9, pp. 79–85, September.

J. Hennessy and D. Patterson, 2011. Computer Architecture: A Quantitative Approach. Morgan Kaufman. Latest edition, e.g., 5th ed.

B. Jacob, S. Ng, and D. Wang, 2007. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann.

K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, 1996. “The case for a single-chip multiprocessor.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 2–11.

A. Smith, 1982. “Cache Memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, August.

D. Sorin, M. Hill, and D. Wood, 2011. A Primer on Memory Consistency and Cache Coherence. Morgan Claypool.