Evaluating Novel Memory System Alternatives for Speculative Multithreaded Computer Systems

AJ KleinOsowski and David J. Lilja

Department of Electrical and Computer Engineering

Minnesota Supercomputing Institute

University of Minnesota

Minneapolis, MN 55455 USA

{ajko, lilja}@ece.umn.edu

Abstract

This work models and evaluates a new cache structure for scalable multithreaded computer systems. Multithreaded architectures which support the speculative execution of multiple concurrent threads of execution require a special speculative memory buffer to detect and potentially correct dependences at runtime. The main question being addressed in this study is whether this speculative memory buffer should be merged with a nonspeculative cache, or kept separate. As a related question, we also evaluate whether the traditional cache structure should be private to each processing element, or whether the cache should be shared among all processing elements. Our results and cost−for−performance analysis show that, on average, the novel hybrid level−1 data cache (which merges a distributed level−1 data cache with the speculative memory buffer) has a 13 percent slowdown as compared to an ideal shared level−1 cache with separate speculative memory buffers. The distributed level−1 cache with separate speculative memory buffer showed, on average, a 4 percent speedup compared to an ideal shared level−1 cache with separate speculative memory buffers.

1. Introduction

Advances in device physics have taken us from the invention of the transistor, to sub−micron feature sizes, to the emerging field of nanoscale electronics. Performance benefits of this feature size scaling, however, are reaching a plateau [1]. In today’s society, high−performance computing permeates every aspect of our lives. No longer are only scientists and engineers interested in squeezing the utmost performance out of their computers. The workloads of today’s mainframe computers, desktop computers, and even handheld devices all can benefit from a performance boost. Since scaling the transistors in these appliances gives only a minimal increase in computing speed, computer architects resort to finding clever ways to compute multiple things at once−−thereby making the workload run faster. Explicitly or automatically detecting and exploiting parallelism in modern workloads shows great potential for performance increases in current and next generation computing devices.

Threads of execution within a program are typically loop iterations or multiple paths of a control structure. These threads most often have cross iteration data dependences that are difficult, if not impossible, for the compiler to detect at compile time. Therefore, multithreaded architectures [6, 8, 14, 18, 21] require hardware to support data dependence checking and speculative execution. In many multithreaded architectures, the compiler identifies possible data dependences and then special hardware determines, at runtime, whether these data dependences are true dependences or simply false alarms. True data dependences are resolved and enforced with speculative memory buffer hardware. False alarms are treated as regular loads and stores, essentially bypassing the speculative memory buffer.

The main question we propose and evaluate in this study is whether this speculative buffer should be merged with a nonspeculative cache, or kept separate. As a related question, we also evaluate whether the traditional cache structure should be private to each processing element, or whether the cache should be shared among all the processing elements. Our motivation is to quantitatively evaluate the performance of a novel, hybrid level−1 data cache, which merges a distributed level−1 data cache with the speculative memory buffer.

In the remainder of this paper, Section 2 describes various multithreaded architectures and how they handle data dependence checking and correcting. Section 2 also describes the motivation for developing a novel, hybrid cache structure. Section 3 describes the Superthreaded Architecture [18] model in detail. Section 4 describes our experimental model and the different configurations we evaluate in this work. Section 5 states the results of our simulations. In Section 6 we present ideas for continuing this work and in Section 7 we conclude with our final recommendation of the best cache structure for multithreaded computer systems.

2. Background and Motivation

Several multithreaded architectures have been proposed which support synchronization and communication between threads. In the M−Machine [6], XIMD [21], Elementary Multithreading [8], and Multiscalar approaches [14], data values are speculated and then, if the speculations turn out to be incorrect, the thread which speculated incorrectly is terminated and restarted. This requires each read and write to memory to be checked and validated. Synchronization information and data are passed among thread units via hardware extensions to the register file. In the case of the Multiscalar approach, another piece of hardware, the speculative versioning cache [7], also can be used to enforce data dependences.

In the Superthreaded [18] approach, data dependences are checked and enforced, not speculated. This means that only writes to memory, not reads, need to be buffered. Since fewer addresses need to be checked in the Superthreaded approach as compared to other multithreaded approaches, the bandwidth required for the Superthreaded speculative memory buffer will be lower than the bandwidth required for other multithreaded speculative memory buffers. Furthermore, the Superthreaded speculative memory buffer will need fewer entries than other multithreaded memory buffers, thereby allowing it to be smaller than other multithreaded memory buffers.

The idea of merging the speculative memory buffer with a distributed traditional cache was first proposed for the Multiscalar architecture [14]. In this approach a bus was used to connect the distributed speculative buffers. Every read and write to a memory address was sent through the local speculative buffer and then broadcast on the bus. A snoopy protocol was used to update the remaining speculative buffers.

Storing information about all the reads and writes to memory requires a very large speculative buffer. Broadcasting all the reads and writes on the bus incurs a large amount of bus traffic. These two factors compound the complexity of the Multiscalar approach which then limits its scalability. If, instead, we assume the Superthreaded [18] approach which requires only writes to be buffered and uses a unidirectional ring for communicating data to downstream speculative buffers, we may be able to better scale the single−chip multithreaded processor with larger caches and more thread units. Our inexpensive hardware approach will also position our architecture as a viable candidate for a multi−chip multithreaded system, as is done by Steffan [16] and Cintra [5].

In this work we use the Superthreaded Architecture [18] as a test platform. Previous work on this architecture assumed a shared level−1 data cache and a distributed speculative memory buffer, separate from the level−1 data cache. Further evaluation of the shared level−1 data cache led to suspicion that threads would contend and stall while waiting for an available read or write port [19]. The only way to resolve this bottleneck would be to have one or more read and write ports per thread unit. On a large scale system with eight thread units, assuming two read ports and one write port per thread unit, this would mean the shared level−1 data cache would need 16 read ports and 8 write ports.

Each port of a multi−ported cache structure is a long wire attached to each bank of the cache. Due to its length, each of these port wires, by itself, adds a tremendous amount of capacitance to the cache structure. When multiple port wires are placed next to one another, their capacitance is compounded by the coupling between the wires. Power consumption is directly proportional to capacitance, so as the capacitance rises, so does the power consumption. Increases in power consumption lead to heat problems which ultimately cause degradation and failure of the chip components. Capacitance effects aside, the die space needed to run multiple wires from each thread unit to the shared level−1 data cache is unrealistically large.

In short, having enough ports on a shared level−1 data cache to reduce the bottleneck effect makes the shared level−1 data cache hardware too expensive to be a viable option. Instead, we choose to distribute the level−1 data cache.

Distributing the level−1 data cache immediately raises a red flag: How do we enforce coherence among the caches? Since our compiler conservatively identifies all possible data dependences in a thread's execution path, these shared data items are kept in the speculative memory buffer. Updates to shared data items are passed to downstream threads via the unidirectional communication ring. In this way, cache coherence is already built into the Superthreaded architecture. Section 3 describes the coherence mechanism of the Superthreaded Architecture in more detail.

Prior work [2] on the Superthreaded architecture showed high variance in the number of active speculative memory buffer entries among different benchmark program workloads. This variance in speculative memory buffer usage means that some programs filled the speculative memory buffer to capacity, and other programs used very few of the speculative memory buffer entries. In the Superthreaded approach, if a speculative thread fills its speculative memory buffer, that thread must stall until all parent threads retire and the thread becomes nonspeculative. Once the thread is nonspeculative, it can flush its speculative memory buffer entries to the main shared memory and continue executing. Needless to say, stalls due to a filled speculative memory buffer serialize execution and drastically increase the runtime of a benchmark program.

In order to avoid stalls due to a filled speculative memory buffer, we propose a hybrid level−1 data cache (shown in Figure 2−1, part c) which merges the speculative memory buffer with a distributed, nonspeculative data cache. Entries in this hybrid structure can be used as either a traditional cache line, or as a speculative memory buffer entry. In this way, we significantly increase the speculative memory buffer entries available to workloads which need many entries. For workloads which need only a few speculative memory buffer entries, the hybrid entries can be used as traditional cache lines, thereby making use of otherwise unused die space.

The hybrid level−1 data cache does not come without tradeoffs. Most notably, in order to avoid overflow in any of the cache rows, the hybrid cache must be able to put an entry in any cache block. This requires a fully−associative lookup when searching the hybrid cache for a block. Also, the coherence mechanism for the Superthreaded Architecture works at the word level. This requires a very small block size of 4 bytes (one word) for the hybrid structure. Our evaluation and the results presented in Section 5 will show whether the increased performance of the hybrid level−1 data cache outweighs the implementation tradeoffs of this novel structure.

Cache Configurations Under Test

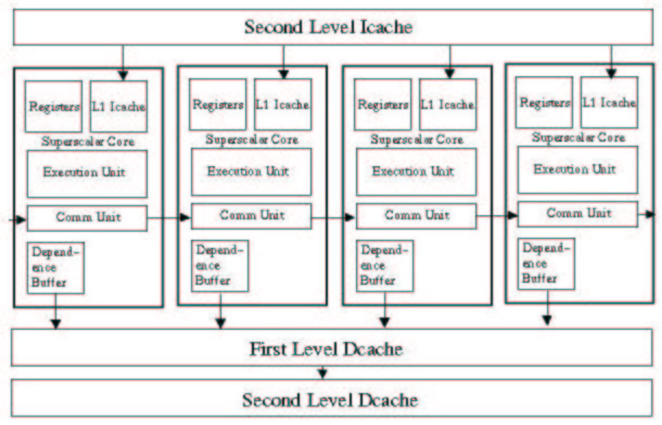

Figure 2−1a. Shared level−1 data cache with distributed, distinct speculative memory buffers

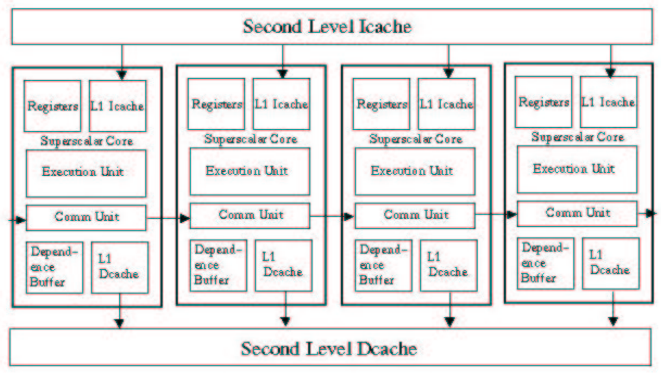

Figure 2−1b. Distributed level−1 data caches with distributed, distinct speculative memory buffers.

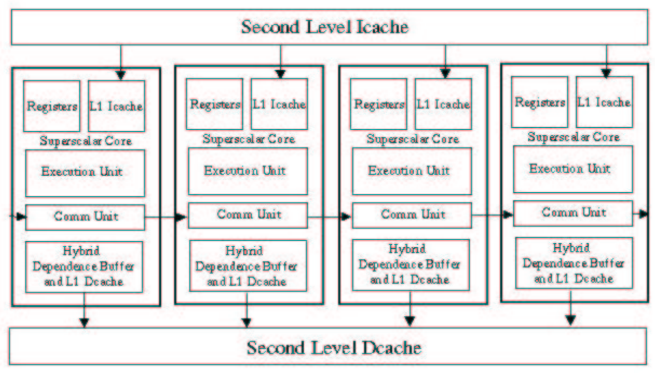

Figure 2−1c. Distributed, hybrid level−1 caches which combine the level−1 data cache and the speculative memory buffer

3. The Superthreaded Architecture Model

The Superthreaded Architecture [18] was developed to address the unexploited parallelism available in classically hard−to−parallelize general purpose programs. As feature sizes shrink, more transistors become available on computer chip dies. The Superthreaded Architecture makes use of these extra transistors by adding special thread management and memory operation filtering hardware to a standard superscalar core. Figure 3−1 shows a thread unit block diagram. (The cache system shown in Figure 3−1 is only one of the three cache systems under investigation in this study.) Multiple copies of the core and special hardware (a so−called thread unit) are then put together on a single die. Figure 3−2 shows the block diagram of four thread units assembled together.

3.1 Superthreaded Hardware

Each thread unit is connected to its successor via a unidirectional communication ring. Thread units each have a local register file and local level−1 instruction cache. The cache arrangement of each thread unit is under investigation in this work.

A program begins executing on a single thread unit while all other thread units are idle. When a parallel region is encountered, a special superthreaded instruction wakes up the downstream thread unit and sends this thread unit all the data it needs to begin executing an iteration of the loop. This newly active thread unit soon executes the instructions to wake up its neighbor and send its neighbor data. Each thread wakes up its downstream thread until there are no more idle threads in the system. The youngest thread stores the wake up instruction and data in a special buffer which holds these instructions until the oldest thread finishes its iteration of the loop and goes back to being idle. At this time, the oldest thread receives the wake up instruction and data and then becomes the youngest thread in the system. This cycle of wake up−execute−idle−wake up proceeds until all iterations of the parallel region are complete.

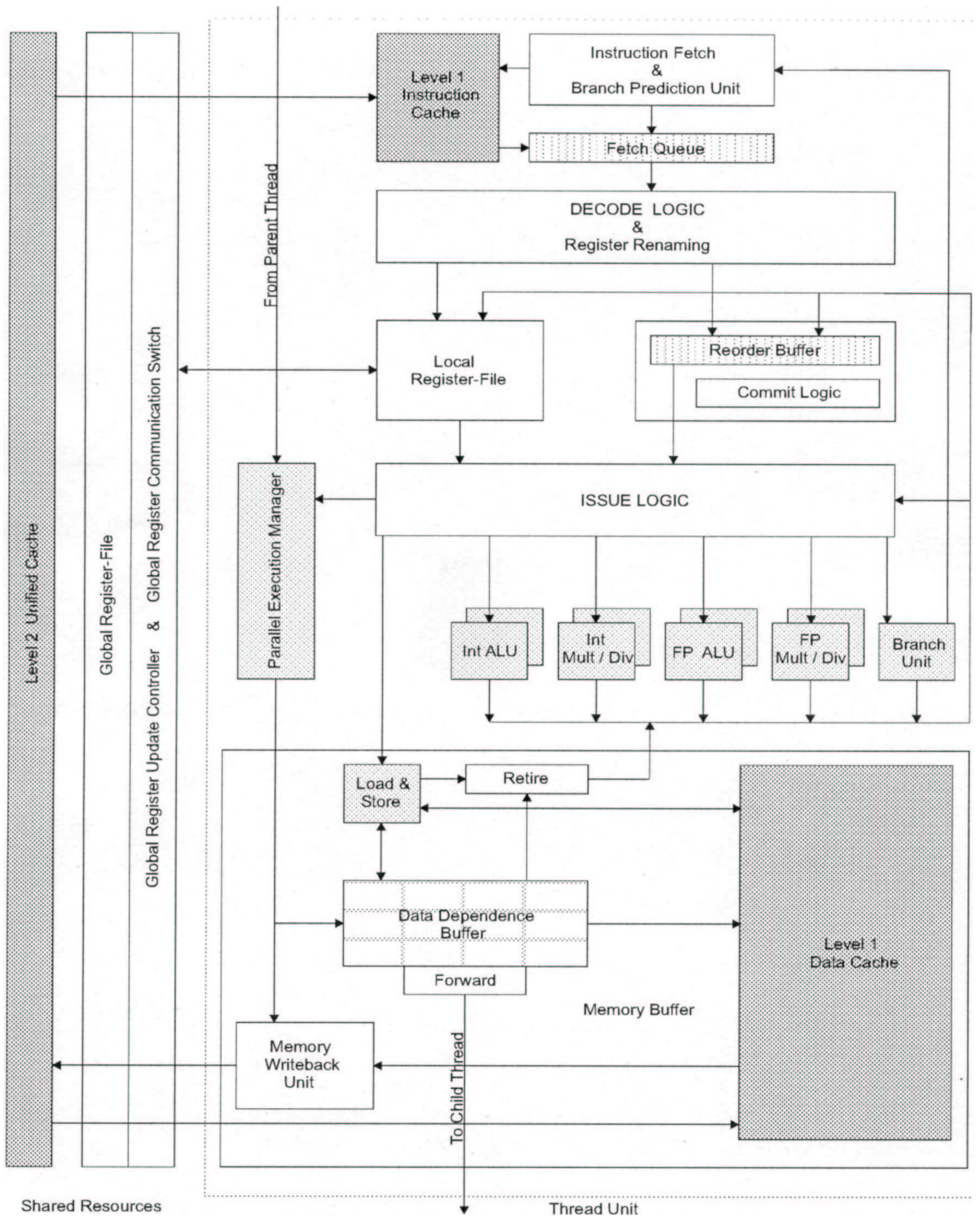

Figure 3−1. Single thread unit block diagram.

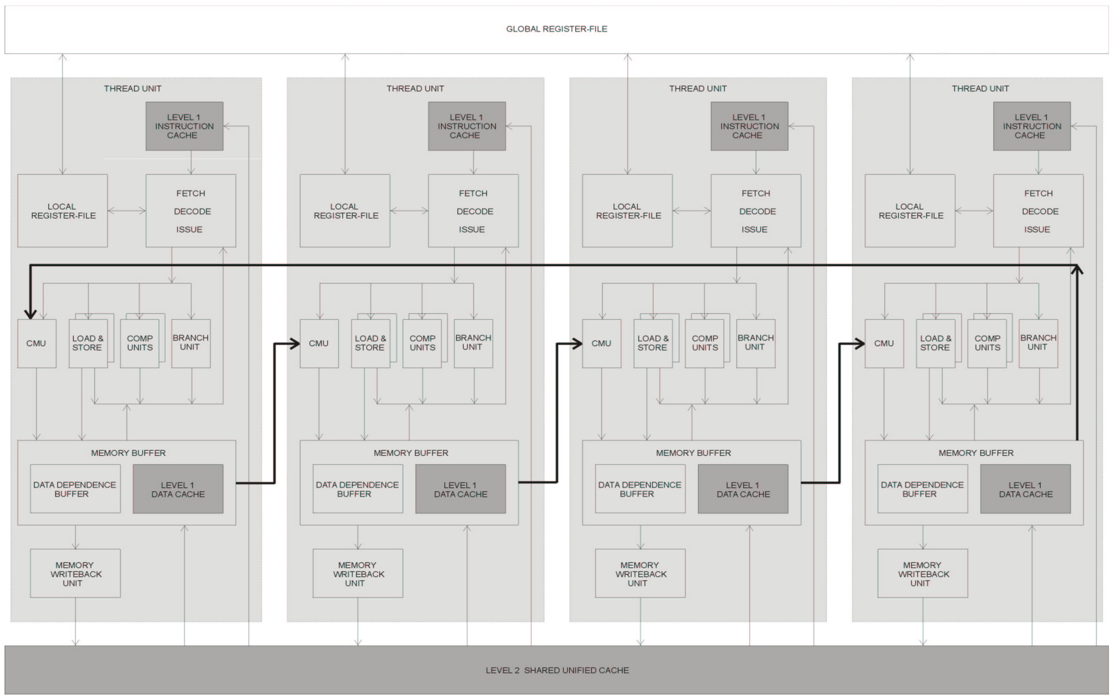

Figure 3−2. Superthreaded Architecture with four thread units.

3.2 Thread Pipelining Execution Model

The thread management hardware, combined with a parallelizing compiler, uncovers and exploits course grained parallelism available in program loops and control structures. The superscalar core and optimizing back end of the parallelizing compiler reorder instructions and exploit instruction level parallelism throughout a program. As the compiler analyzes code, it applies a series of source−to−source transformations which identify the parallel regions of code and instruct the hardware when and how to communicate amongst the thread units.

This so−called Thread Pipelining Execution Model allows for maximum overlap among threads while preserving the semantics necessary for correct program execution. Figure 3−3 pictorially shows the stages of the thread pipelining execution model. The sections below describe the purpose and function of each stage.

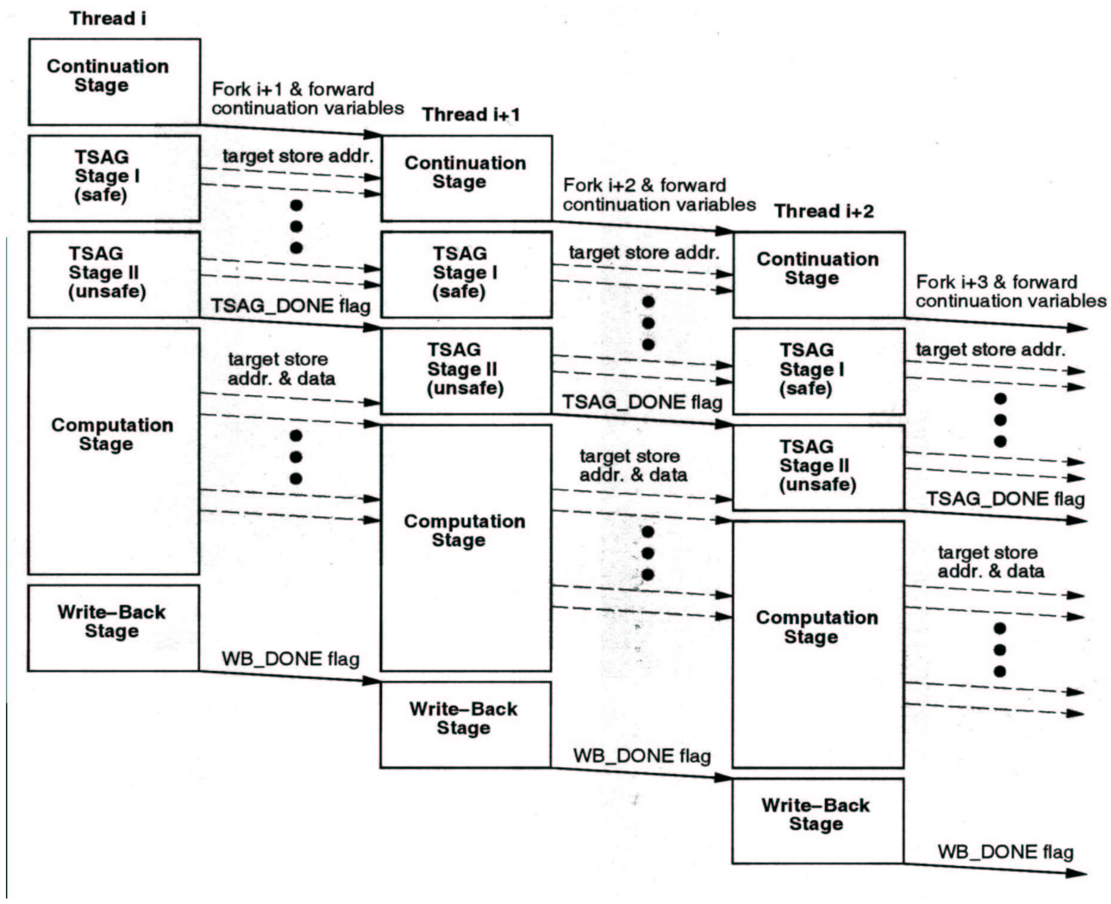

Figure 3−3. Thread pipelining execution model.

<u>*Continuation Stage*</u>

The primary task of the continuation stage is to calculate and send loop index values to the downstream (child) thread. The loop index counter determines whether or not an iteration is needed, so calculating and sending this value early is essential for starting downstream threads early. The continuation stage begins with the ST_BEGIN instruction and finishes when the actual FORK instruction is executed and sent to the downstream thread. When the downstream thread receives a FORK instruction, it will already have all the continuation variables needed to check its entry conditions.

Target Store Address Generation Stage

The target store address generation stage, commonly abbreviated as the TSAG stage, is used to flag all possible maybe dependences. In particular, if there is any chance an address may be written by multiple threads, an entry is allocated for that address in the local memory buffer and the address is sent to all downstream threads. If these downstream threads encounter a read from this address, they must wait for the value from an upstream thread, rather than read the value from memory.

The TSAG stage is divided into two sub−stages. The first sub−stage, the safe TSAG stage, consists of allocations where the address is independent of any calculations performed by upstream threads. If an address is the result of a calculation performed by an upstream thread, that allocation must occur in the unsafe TSAG stage. The unsafe TSAG state is bracketed by synchronization instructions which tell the thread hardware to wait for the result of the upstream thread's address calculation before completing the allocation.

Computation Stage

All of the useful calculations of a loop iteration occur in the computation stage. During computation, every memory address read is checked against the addresses and values in the memory buffer. If an address is in the memory buffer but no valid data is in the memory buffer for that address, the instruction stalls until valid data arrives from an upstream thread. In this way, cross−iteration dependences are enforced. Since the Superthreaded architecture contains an out−of−order superscalar core, even though one instruction is stalled, computation may continue on other in−flight instructions which are not dependent on the stalled instruction. The computation stage terminates when a thread processes the ST_END instruction, signaling the end of a loop iteration.

Write Back Stage

Only the non−speculative head thread can write to memory. Therefore, after a thread processes the ST_END instruction, it goes idle until all upstream threads have retired and it becomes the head thread. At this time, the thread begins transferring the values in its memory buffer to the shared memory. Upon completion of this write back operation, the thread sends a Writeback_Done signal to the downstream (child) thread, thereby making the downstream thread the new head thread.

When a thread finishes its continuation, TSAG, computation, and writeback stages, the thread goes idle. This newly idle thread is now the youngest thread in the system and is ready to ready to receive the continuation variables and FORK instruction queued in the forwarding queue of the now second−to−youngest thread in the system. This round−robin progression of continuation−TSAG−computation−writeback−idle−receive continuation information proceeds in each thread unit throughout the parallel region of a program. A parallel region terminates when the last active thread finishes its writeback stage. That active thread then continues executing the sequential portion of the program, rather than going idle.

3.3 Compiling Superthreaded Code

Agassiz [23], the parallelizing compiler for the Superthreaded architecture, is still under development. Therefore, we hand−parallelized benchmark codes for use in our simulations. Superthreaded instructions are inserted into the code as function calls. These function calls are transformed to jump−to−label instructions in the assembly code by our development compiler, then these jump−to−label instructions are transformed to the actual superthreaded instructions by a backpatching pass of the development compiler. Figure 3−5, parts a and b, show an example of a loop which has been transformed with the Superthreaded source code transformations.

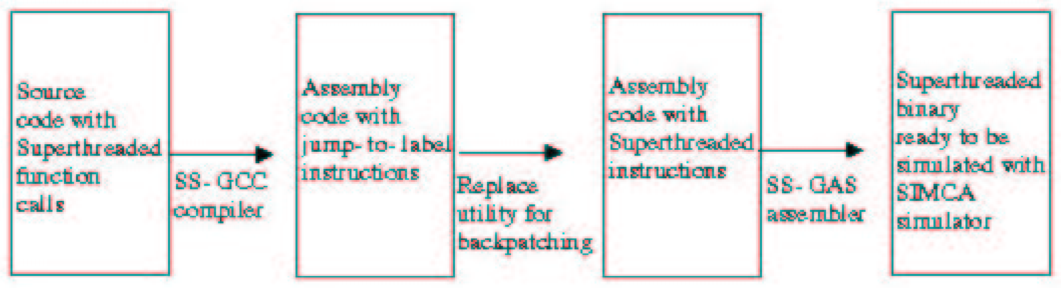
Source code with Superthreaded function calls
SS-GCC compiler
Assembly code with jump-to-label instructions
Replace utility for backpatching
Assembly code with Superthreaded instructions
SS-GAS assembler
Superthreaded binary ready to be simulated with SIMCA simulator

Figure 3−4. Development compiling process for Superthreaded code.

Figure 3−4 pictorially shows the compilation process for this transformed code. Once code is compiled, the Superthreaded binary is simulated using Simca [9], the cycle accurate functional and timing simulator.

```
while ( funct_units[i].class != ILLEGAL_CLASS) {
     if (f->class == funct_units[i].class) {
          if (minclk > funct_units[i].busy) {
               minclk = funct_units[i].busy;
               j = i;
               if (minclk == 0) break;
          }
     }
     i++;
}
```

Figure 3−5a. An example code segment taken from the 124.m88ksim benchmark, from the SPEC 1995 suite.

```
/* Continuation Stage */
L1:
     ST_BEGIN();
     i_1 = i;
     STORE_TS(&i, i_1+1);
     FORK();
/* Target Store Address Generation Stage */
     ALLOCATE_TS(&minclk);
     WAIT_TSAG_DONE();
     /* no unsafe TSAG in this example */
     RELEASE_TSAG_DONE();
/* Computation Stage */
     if (funct_units[i_1].class == ILLEGAL_CLASS) {
          ABORT_FUTURE();
          i = i_1;
          goto L2;
     }
     if (f->class == funct_units[i_1].class) {
          if(minclk > funct_units[i_1].busy) {
               STORE_TS(&minclk,
funct_units[i_1].busy);
               j = i_1;
               /*if minclk is zero, break to
terminate*/
               if (minclk == 0) {
                    ABORT_FUTURE();
                    i = i_1;
                    goto L2;
               }
          }
          else
               RELEASE_TS(&minclk);
     }
     else
          RELEASE_TS(&minclk);
     ST_END();
/* Writeback stage performed automatically by the
hardware */
/* End of Superthreaded parallel region */
L2:
```

Figure 3−5b. The Superthreaded code corresponding to the code segment in part a.

4. Methodology

In this study, we simulate the functionality and performance of three cache architectures for speculative multithreaded computer systems. To that end, we simulate the access time and power requirements of generic cache architectures of the same sizes, associativities, and port requirements of those used in our functional simulations. We then extrapolate cycle latencies from the access time results (given in nanoseconds) and feed these cycle latencies into our functional simulations to obtain overall performance results.

Each of the three cache architectures we evaluate has its own requirements in terms of associativity and block size. Taking the access time and power requirements of these different cache architectures into consideration in the functional simulations helps us rigorously evaluate which cache architecture alternative is the best in terms of implementation cost and functional performance for the Superthreaded Architecture.

4.1 Access Time and Power Requirements Simulation

We use the Cacti [20] simulator to gather access time estimates in nanoseconds and per−access power usage in nanojoules. Cacti uses an analytical model based on transistor and wire characteristics derived from SPICE [3] simulations. With Cacti, we can vary the block size, number of ports, number of sets, associativity, and fabrication technology parameters.

In this work we assume a target machine fabricated with 0.18 micrometer fabrication technology running at a clock speed of 1Ghz. This means a cache with a 1 nanosecond access time would have a 1 cycle latency. To convert the nanosecond access times from the Cacti simulations to cycle latencies for the performance simulations, we use the formula

$$\textit{Cache Latency} = \textit{ceiling}\left(\frac{\text{Access Time}}{\text{1 nanosecond}}\right)$$

Per−access power consumption is calculated using an empirical formula built into to Cacti. Cacti's power model takes into consideration the technology parameter, the length of the port wires based on the cache size, and the amount of hardware needed to drive varying levels of cache associativity.

4.2 Functional Simulation

The three cache architectures which we functionally test, and the nomenclature we use to refer to these configurations is:

S (shared):	shared level−1 data cache with distributed, distinct speculative memory buffers
D (distributed):	distributed level−1 data cache with distinct, distributed speculative memory buffers
H (hybrid):	distributed, hybrid level−1 data cache which combines the level−1 data cache and the speculative memory buffer

Figure 2−1 pictorially shows these three configurations.

Simulations are performed with our execution driven simulator, Simca [9], which is an extension of the sim−outorder simulator from the SimpleScalar [4] suite. For our baseline performance, we simulate a shared level−1 data cache configuration with infinite read and write ports, yet for the access time of this cache structure, we use the access time of a structure with two read ports and one write port. This infinite port assumption and overly optimistic access time will make the performance of the shared cache appear better than it could ever achieve on a real system. Thus, these baseline simulations show an upper bound on the performance that can be obtained on this multithreaded architecture. Comparing the results obtained with the distributed and hybrid configurations to this upper bound then shows how much performance we may be sacrificing in the worst case by switching to one of these lower−cost implementations.

To compare the performance of the three cache configurations, we run three sets of simulations, each with a different cache size. Prior sensitivity analysis of the Superthreaded Architecture [18] gave us a starting point for the cache size for the Group 1 simulations. The cache sizes in the Group 2 simulations are twice as large as the cache sizes in the Group 1 simulations, and the cache sizes in the Group 3 simulations are twice as large as the cache sizes in the Group 2 simulations. Doubling and then quadrupling the cache sizes will give us a good indication of if our performance trends hold as we scale up the computer system.

We keep the hardware cost of the different configurations within a group approximately equal. For example, for the first set of simulations, we use a 32KB shared cache with four thread units. This 32KB shared cache is equivalent to four 8KB distributed caches. For the shared and distributed cache configurations, we use a 128 entry (512 byte) speculative memory buffer. Once again, 128 entries was chosen as an appropriate size based on prior sensitivity analysis of the memory buffer [2] in the Superthreaded Architecture. In the case of the hybrid configuration, we do not have the additional hardware cost of the 128 entry speculative memory buffer, since the hybrid cache is, in essence, a very large, general purpose, speculative memory buffer. However, the hybrid configuration is fully associative whereas the shared and distributed configurations are 4−way set associative. The added overhead of the fully associative cache decoder is approximately equivalent to the missing 512 bytes incurred by the distinct speculative memory buffer, thereby making all three configurations approximately equal in terms of hardware cost.

Sensitivity analysis performed in another prior work [2] showed that the Superthreaded Architecture performed best with four or eight thread units. With this in mind, we run our simulations first with four thread units, then again with eight thread units.

Table 4−1 summarizes the configuration parameters used for each simulation. The latencies for these structures are determined by the cache access time simulations run with Cacti. These latencies will be stated and discussed in Section 5.

As our workload, we use a combination of programs from the SPEC 92, SPEC 95, and SPEC2000 suites [15], as well as a few standard unix utilities. To obtain results in a reasonable amount of time, we used reduced reference input files [11] for some benchmarks and truncated the simulation of other benchmarks. Table 4−2 lists our benchmark programs, their origins, their committed instruction counts, and their total number of accesses to the level−1 data cache.

Simulation Name	***Parameters***
Group 1:	
1D{4,8}	Distributed level−1 data cache, 8KB, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
1S{4,8}	Shared level−1data cache, 32KB for 4 thread units, 64KB for 8 thread units, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
1H{4,8}	Hybrid level−1 data cache, 8KB, 4 bytes/block, fully associative, random replacement policy
Group 2:	
2D{4,8}	Distributed level−1data cache, 16KB, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
2S{4,8}	Shared level−1data cache, 64KB for 4 thread units, 128KB for 8 thread units, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
2H{4,8}	Hybrid level−1 data cache, 16KB, 4 bytes/block, fully associative, random replacement policy
Group 3:	
3D{4,8}	Distributed level−1data cache, 32KB, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
3S{4,8}	Shared level−1data cache, 128KB for 4 thread units, 256KB for 8 thread units, 32 bytes/block, 4−way associative, 128 entry (512 byte) speculative memory buffer, random replacement policy, 4 or 8 thread units
3H{4,8}	Hybrid level−1 data cache, 32KB, 4 bytes/block, fully associative, random replacement policy

Table 4−1. Simulation nomenclature and parameters used for each simulation. All simulations use a 2MB, 64 bytes/block, 4−way associative level−2 data cache. Simulations are run first with 4 thread units, then with 8 thread units.

Benchmark	***Source***	***Instructions Simulated (in millions)***	***Level−1 Data Cache Accesses (in millions)***
Alvinn	SPEC FP 1992	94.3	35.8
Cmp	Standard Unix Utility	5.8	1.6
Compress	SPEC INT 1995	1.9	0.6
Ear	SPEC FP 1992	1020.2	282.7
Hydro2d	SPEC FP 1992	33.4	8.6
M88k	SPEC INT 1995	1063.2	305.7
Mcf	SPEC INT 2000	1619.3	720.2
Wc	Standard Unix Utility	6.0	2.0

Table 4−2. Benchmark programs used in this study.

5. Results

The following results show the implementation costs and performance results of our three alternate cache architectures for multithreaded computer systems.

5.1 Power Usage

Ideal performance of our shared level−1 data cache would require a large number of ports. Therefore, we began our analysis by looking at the power implications of scaling the number of read and write ports on the shared level−1 cache.   Figure 5−1 shows these results for a 128KB cache with 32 bytes per block and 4−way associativity.

In Figure 5−1, we see that the plot of per−access power usage follows a quadratic curve. This result confirms our assumption that adding enough ports (two read ports and one write port per thread) to the shared level−1 data cache to avoid contention would make the shared structure unrealistically expensive (in terms of power) to implement.

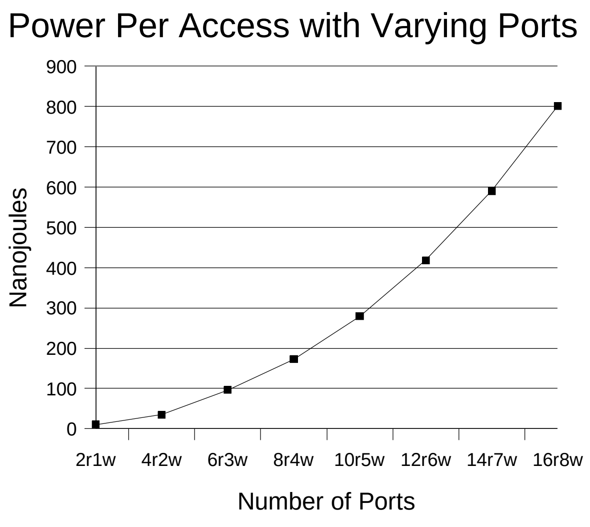

Figure 5−1. Power per access for shared 128KB level−1 data cache with varying number or read and write ports.

Table 5−1 shows the per−access power usage of the different configurations studied in the performance analysis simulations. From these results, we see that the distributed level−1 cache with the distinct memory buffer has the lowest per−access power usage across all three groups of simulations. We also see that the shared level−1 cache has an exorbitant per−access power usage. In the case of the group 3 simulations, the per−access power usage of the shared level−1 cache configuration is 3051 percent higher than the per−access power usage of the distributed level−1 cache configuration. In all groups of simulations, the per−access power usage of the hybrid cache configuration fell somewhere between the per−access power usage of the distributed and shared level−1 cache configurations.

Since the per−access power usage of the shared level−1 cache configuration in Table 5−1 was so high compared to the power usage of the distributed and hybrid cache configurations, we proceeded to look at how that power usage scaled down if we reduced the number of ports on the shared cache. Table 5−2 compares the results of 2 read ports and 1 write port *per thread*, the ideal performance case, to the ideal power case where we have only 2 read ports and 1 write port, regardless of the number of threads. Wilson and Olukotun [19] concluded that the best performance per implementation cost occurred with two cache ports. Therefore, in Table 5−2 we also include a column showing the per−access power usage of a compromise case with 4 read ports and 2 write ports, regardless of the number of threads.

In the ideal power case, where all cache structures have 2 read ports and 1 write port, we see that the power use of the shared level−1 cache configuration is the lowest in all simulations by at least 35%. However, when we look at the compromise case, we see that the distributed level−1 cache once again has the lowest power usage in all but one simulation. Even in the eight thread group 1 simulations, where the shared level−1 cache configuration used less power than the distributed level−1 configuration, the power savings were only 20% of the distributed level−

1 configuration.

All simulations studied in the performance simulations use a 128 entry memory buffer. Rather than introduce novel new cache architectures, the naïve way to avoid memory buffer overflow is to increase the size of the memory buffer. Table 5−3 shows the power implications of increasing the memory buffer from 128 entries, to 256 entries, to 512 entries. In this table, we note that the same trends in power usage apply across all columns. Namely, the distributed level−1 cache configuration has the lowest per−access power usage, the shared level−1 cache configuration has the highest per−access power usage, and the hybrid cache configuration's power usage falls somewhere between the power usage of the distributed and shared level−1 cache configurations.

Configuration	MB Power (nJ)	L1 Power (nJ)	Total Power (nJ)
1d4	4*1.61=6.44	4*3.88=15.52	21.96
1s4	4*1.61=6.44	58.99	65.43
1h4	4*11.93=47.72	0.00	47.72
1d8	8*1.61=12.88	8*3.88=31.04	43.92
1s8	8*1.61=12.88	463.06	475.94
1h8	8*11.93=95.44	0.00	95.44
2d4	4*1.61=6.44	4*4.37=17.48	23.92
2s4	4*1.61=6.44	101.67	108.11
2h4	4*21.0=84.00	0.00	84.00
2d8	8*1.61=12.88	8*4.37=34.96	47.84
2s8	8*1.61=12.88	800.86	813.74
2h8	8*21.0=168.00	0.00	168.00
3d4	4*1.61=6.44	4*5.17=20.68	27.12
3s4	4*1.61=6.44	172.85	179.29
3h4	4*39.11=156.44	0.00	156.44
3d8	8*1.61=12.88	8*5.17=41.36	54.24
3s8	8*1.61=12.88	1642.02	1654.90
3h8	8*39.11=312.88	0.00	312.88

Table 5−1. Power per access for configurations used in the performance analysis.

Configuration	8r4w, 16r8w Ports Ideal Performance Total Power (nJ)	2r1w Ports Ideal Power Total Power (nJ)	4r2w Ports Compromise Total Power (nJ)
1d4	21.96	21.96	21.96
1s4	65.43	11.61	22.04
1h4	47.72	47.72	47.72
1d8	43.92	43.92	43.92
1s8	475.94	19.87	35.08
1h8	95.44	95.44	95.44
2d4	23.92	23.92	23.92
2s4	108.11	13.43	28.64
2h4	84.00	84.00	84.00
2d8	47.84	47.84	47.84
2s8	813.74	23.19	48.01
2h8	168.00	168.00	168.00
3d4	27.12	27.12	27.12
3s4	179.29	16.75	41.57
3h4	156.44	156.44	156.44
3d8	54.24	54.24	54.24
3s8	1654.90	34.27	87.62
3h8	312.88	312.88	312.88

Table 5−2. Power per access with shared level−1 configurations with varying numbers of ports.

Configuration	128 MB Entries Total Power (nJ)	256 MB Entries Total Power (nJ)	512 MB Entries Total Power (nJ)
1d4	21.96	27.04	31.76
1s4	65.43	70.51	75.23
1h4	47.72	47.72	47.72
1d8	43.92	54.08	63.52
1s8	475.94	486.10	495.54
1h8	95.44	95.44	95.44
2d4	23.92	29.00	33.72
2s4	108.11	113.19	117.91
2h4	84.00	84.00	84.00
2d8	47.84	58.00	67.44
2s8	813.74	823.90	833.34
2h8	168.00	168.00	168.00
3d4	27.12	32.20	36.92
3s4	179.29	184.37	189.09
3h4	156.44	156.44	156.44
3d8	54.24	64.40	73.84
3s8	1654.90	1665.06	1674.50
3h8	312.88	312.88	312.88

Table 5−3. Power per access with varying numbers of memory buffer entries.

5.2 Access Time Requirements

The analysis of the access time results very closely parallels the analysis of the power usage results. We see in Figure 5−2 that when we scale up the number of ports for the shared level−1 cache configuration to the point where we have 2 read and 1 write port per thread, the access time increases quadratically. At the high end of this curve, we see an access time of 31.15 nanoseconds. Assuming a 1GHz target machine, this equates to a level−1 cache latency of 32 cycles. Most commercial processors have a 1 or 2 cycle latency for their level−1 data cache. By direct comparison, we see that implementing a shared level−1 data cache with enough ports to avoid contention would actually slow overall performance.

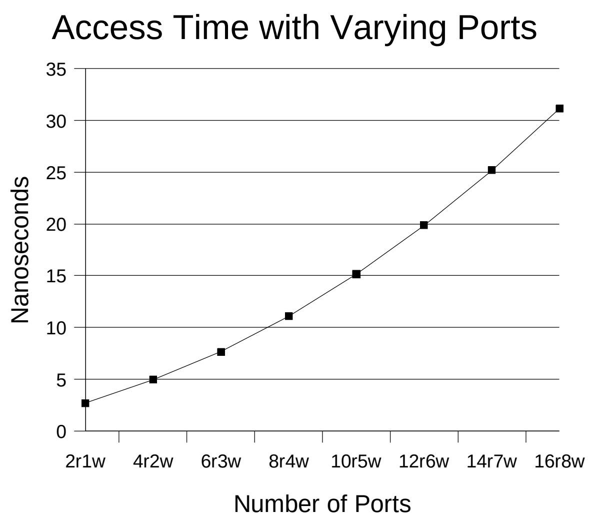

Figure 5−2. Access time for shared 128KB level−1 data cache with varying number or read and write ports.

In Table 5−4, we see the access times of the various configurations used in the performance analysis. These access times are converted into cycle latencies (assuming a 1GHz target machine) and then fed as parameters to our performance simulations. We use the shared level−1 cache configuration as our baseline for our performance simulation. Therefore, we use the best access times possible for the shared level−1 data cache. These best access times occur with 2 read ports and 1 write port. Table 5−5 shows how the shared level−1 data cache access time varies when we scale up the number of ports.

We see in Table 5−4 that the distributed level−1 data cache configuration has the fastest access time. Despite this fast access time, we cannot assume that the distributed level−1 data cache configuration will have better performance than the shared or hybrid configurations. The 128 entry memory buffer used with the distributed configuration may overflow and cause serial execution of the parallel regions, thereby making the hybrid configuration the best performer. Since the distributed level−1 caches are one−fourth or one−eighth the size of the shared level−1 cache, we may experience higher miss rates with the distributed configuration, thereby making the shared configuration the best performer. The actual performance results gathered from the access times stated in Table 5−4 will be discussed in the next section.

In Table 5−6, we see the access time impact of increasing the memory buffer size. Counter to our intuition, we see that increasing the memory buffer entries to 256 (from 128 entries) brings the access time down, not up. The access time from 128 entries to 512 entries goes up, as our intuition tells us it would. We attribute this non−intuitive access time of the 256 entry memory buffer to round off or other errors in Cacti.

Configuration	128 MB Entries MB Access (ns)	2r1w Ports Ideal Performance L1 Access (ns)
1d4	1.95	1.24
1s4	1.95	1.70
1h4	2.48	2.48
1d8	1.95	1.24
1s8	1.95	2.05
1h8	2.48	2.48
2d4	1.95	1.43
2s4	1.95	2.05
2h4	3.24	3.24
2d8	1.95	1.43
2s8	1.95	2.69
2h8	3.24	3.24
3d4	1.95	1.70
3s4	1.95	2.69
3h4	4.80	4.80
3d8	1.95	1.70
3s8	1.95	3.27
3h8	4.80	4.80

Table 5−4. Access time for configurations used in the performance analysis.

Configuration	8r4w, 16r8w Ports Ideal Performance L1 Access (ns)	2r1w Ports Ideal Performance L1 Access (ns)	4r2w Ports Compromise L1 Access (ns)
1d4	1.24	1.24	1.24
1s4	5.35	1.70	2.61
1h4	2.48	2.48	2.48
1d8	1.24	1.24	1.24
1s8	19.39	2.05	3.38
1h8	2.48	2.48	2.48
2d4	1.43	1.43	1.43
2s4	7.16	2.05	3.38
2h4	3.24	3.24	3.24
2d8	1.43	1.43	1.43
2s8	31.15	2.69	4.95
2h8	3.24	3.24	3.24
3d4	1.70	1.70	1.70
3s4	11.08	2.69	4.95
3h4	4.80	4.80	4.80
3d8	1.70	1.70	1.70
3s8	45.38	3.27	6.27
3h8	4.80	4.80	4.80

Table 5−5. Access time with shared level−1 configurations with varying numbers of ports.

128 MB Entries MB Access (ns)	256 MB Entries MB Access (ns)	512 MB Entries MB Access (ns)
1.95	1.82	2.03

Table 5−6. Access time with varying numbers of memory buffer entries.

5.3 Performance Results

The performance results in Figure 5−3 show that the distributed configuration has, on average, a 4 percent speedup compared to the ideal shared configuration. The hybrid configuration has, on average, a 13 percent slowdown compared to the ideal shared configuration.

The distributed configuration was faster than the ideal shared configuration in 39 of the 48 simulations. The worst performance of the distributed configuration was for the group 1, four thread simulation of Hydro2d with a 3.24 percent slowdown as compared to the ideal shared configuration. The best performance for the distributed configuration was the group 3, eight thread simulation of compress with a 15.37 percent speedup. Overall, the group 1, four thread simulations of the distributed configuration showed minimal slowdown while all other simulations (except for the group 2, eight thread simulation of mcf) showed positive speedup.

The hybrid configuration was slower than the ideal shared configuration in all 48 simulations. The best performance of the hybrid configuration was the group 1, eight thread simulation of alvinn with a 0.22 percent slowdown. Overall, alvinn had only modest slowdown for the hybrid configuration with an average slowdown of 2.03 percent. Most other simulations (35 of the 48), however, had greater than 10 percent slowdown.

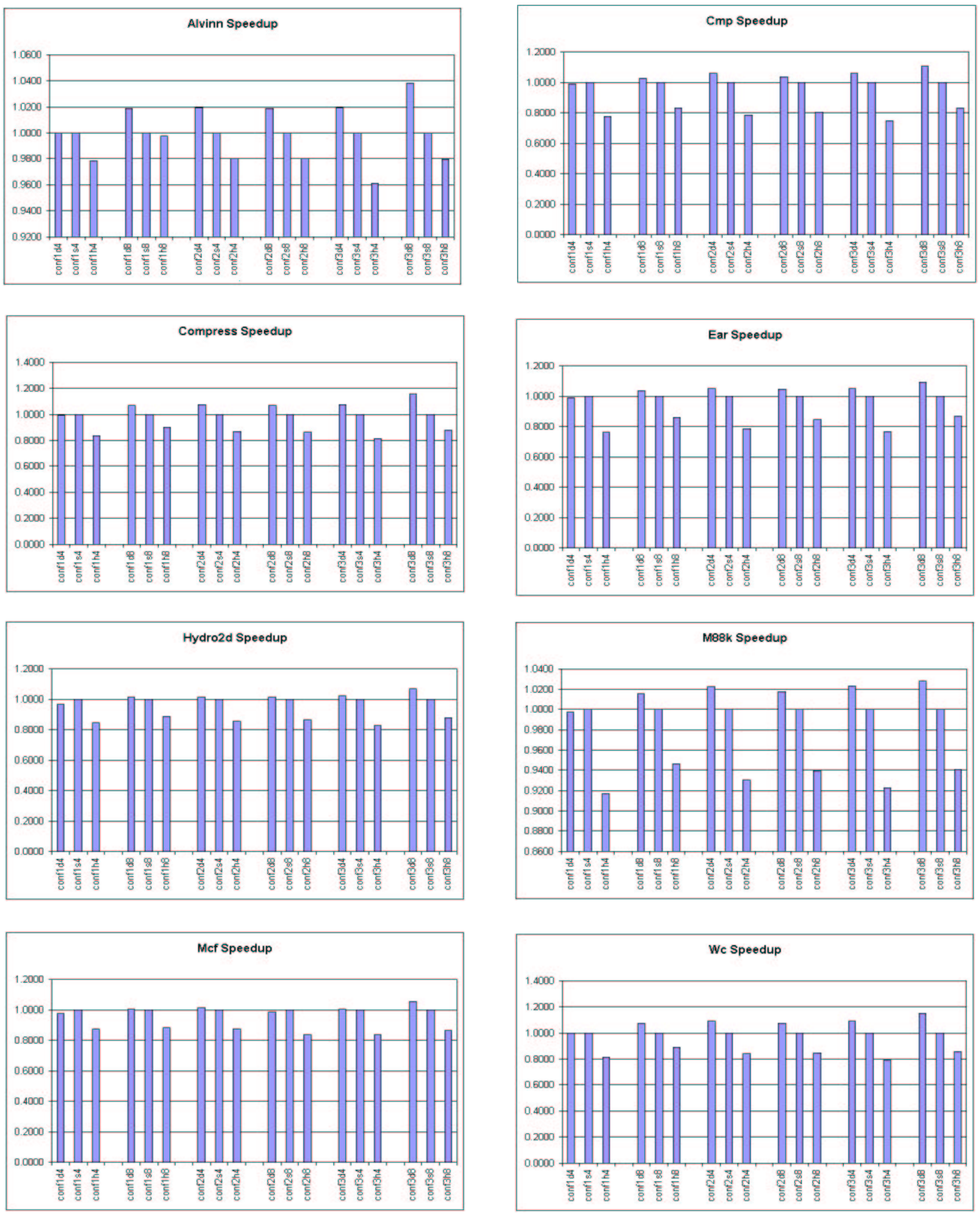

Figure 5−3. Performance results. Runtime is normalized per group to the runtime of the shared configuration.

6. Future Work

Our speculative memory buffer currently uses one word per cache line. This small granularity results in a large amount of transistor overhead for the tag array, as compared to the amount of data stored in the cache. In future work, we plan to develop a technique to increase the granularity of the cache lines in the speculative memory buffer to a more realistic size.

As we scale the speculative memory buffer to sizes greater than 128 entries, the access time becomes very long, as shown in Table 5−6. In order to offset the long access time for large speculative memory buffers, we plan to evaluate the memory access patterns in our benchmarks and determine if we can make the speculative memory buffer 16−way or 32−way associative [22] without causing overflow in the memory buffer rows.

In order to test our Superthreaded Architecture with coarse−grained parallelism, we also would like to build off the work of Cintra [5] and Steffan [16] and develop a simulator for a system of multiple Superthreaded chips with a hierarchy of fine−grained and coarse−grained speculation.

7. Conclusion

In this work, we set out to study whether the speculative memory buffer should be merged with a nonspeculative cache, or kept separate. As a related question, we also evaluated whether the traditional cache structure should be private to each processing element, or whether the cache should be shared among all processing elements.

The shared cache is an attractive option since you can amortize the die space necessary for the cache across the thread units and thereby make the cache very large. However, since this large cache will be far away from the thread units, the access time will be high due to the long port wires between the cache and the thread units. Also, multiple thread units will contend for access to the cache each cycle, inducing stalls.

The distributed cache could be an alternative to the shared cache since the small distributed cache is local to each thread, thereby eliminating port contention and long port wires. However, the distributed caches must be small since we have multiple copies of the cache. The small cache will not be able to hold many cache lines, and therefore may have frequent misses.

Both the shared and distributed caches have the speculative memory buffer in a small, separate structure. If this structure overflows, threads are halted and otherwise parallel execution becomes serial. The hybrid cache eliminates the overflow problem by allowing every line in the cache to act as a speculative memory buffer entry. The tradeoff for the hybrid cache is that the hybrid cache must be fully associative in order to allow a cache block to be placed anywhere in the cache. As the hybrid cache scales in size, its access time becomes large due to the fully associative nature of this structure. Also, the block size of the hybrid cache must be very small (4 bytes) in order to work with existing coherence policies.

Since the performance of the hybrid configuration was worse than that of the ideal shared configuration in all simulations, our results show that we did not have enough instances of memory buffer overflow to outweigh the increased access time of the hybrid configuration. This means we are better off with a distinct speculative memory buffer in its own small structure. This small structure may overflow at times, however the fast access time of this small structure will make up for the performance loss the few times it overflows. A general purpose level−1 data cache should be used in conjunction with this special hardware structure.

Our results show that distributing the level−1 data cache over the thread units improves performance over the large shared level−1 configuration in most cases. In the cases where there is a slowdown, the slowdown is very modest. Implementing a shared level−1 data cache with a large number of read and write ports would be unrealistically expensive in terms of die size and power consumption. Therefore, since our distributed level−1 data cache has much lower power consumption and much faster access time, multithreaded computer systems should make use of small, per−thread level−1 data caches and separate, small, specialized structures for handling cross−iteration thread dependences.

Acknowledgments

This work was supported in part by National Science Foundation grants EIA−9971666 and CCR−9900605, by a University of Minnesota−IBM Shared Research Project equipment grant, and by the Minnesota Supercomputing Institute. Many thanks to the attendees of the Workshop for Memory Performance issues (ISCA 2001) for their helpful feedback on the preliminary version [12] of this work. Thanks, as well, to Chris Johnson [10] and Chuck Li [13] for their work on the VHDL and Verilog implementations of the memory structures examined in this work.

References

1. V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," International Symposium on Computer Architecture (ISCA), June 2000.

2. C. Amlo, "The Superthreaded Multiprocessor: The Instruction Set Architecture and the Parallel Execution Manager", Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 99−08, September, 1999. (MS Thesis) Paper available at http://www.arctic.umn.edu.

3. W. Banzhaf, <u>Computer−Aided Circuit Analysis Using SPICE</u>, Prentice Hall, 1989.

4. D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 3.0," University of Wisconsin −−Madison Computer Sciences Department. Distribution web page located at http://www.simplescalar.org.

5. M. Cintra, J. Martinez, J. Torrellas, "Architecture Support for Scalabel Speculative Parallelization in Shared−Memory Multiprocessors," International Symposium on Computer Architecture (ISCA), June 2000.

6. M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, W. Lee, "The M−Machine Multicomputer," International Symposium on Microarchitectures (MICRO), November 1995.

7. S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative Versioning Cache," International Symposium for High−Performance Computer Architecture (HPCA), 1998.

8. H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," International Symposium on Computer Architecture (ISCA), May 1992.

9. J. Huang and D. Lilja, " An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture," Proceedings of the 6th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, July, 1998.

10. C. Johnson, "Superthreaded Multi−processor: Hardware Design of the Memory Buffer Unit," MS Thesis, Department of Electrical and Computer Engineering, University of Minnesota, October, 1999.

11. A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja, "Adapting the SPEC 2000 Benchmark Suite for Simulation−Based Computer Architecture Research," Workshop on Workload Characterization, International Conference on Computer Design (ICCD), 2000.

12. A KleinOsowski, D. Lilja, "Performance Analysis of a Novel Cache Architecture for Speculative Multithreaded Computer Systems," Workshop on Memory Performance Issues, International Symposium for Computer Architecture (ISCA), 2001.

13. C. Li, "Memory Buffer Implementation in the Superthreaded Architecture", MS Thesis, Department of Electrical and Computer Engineering, University of Minnesota, August, 2001.

14. G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," International Symposium on Computer Architecture (ISCA), June 1995.

15. SPEC Benchmark Suite. Information available at http://www.spec.org.

16. J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A Scalable Approach to Thread−Level Speculation," International Symposium on Computer Architecture (ISCA), June 2000.

17. J. Torrellas, L.Yang, A. Nguyen, "Toward a Cost−Effective DSM Organization That Exploits Processor−Memory Integration," International Symposium on High Performance Computer Architecture (HPCA), January 2000.

18. J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew, "The Superthreaded Processor Architecture," IEEE Transactions on Computers, Volume 48, Number 9, September 1999, p. 881−902.

19. K. Wilson and K. Olukotun, "High Bandwidth On−Chip Cache Design," IEEE Transactions on Computers, Volume 50, Number 4, April 2001, p. 292−307.

20. S. Wilton and N. Jouppi, "Cacti: An enhanced cache access and cycle time model," IEEE Journal of Solid−State Circuits, May 1996. Tool available at http://www.research.compaq.com/wrl/people/jouppi/CACTI.html.

21. A. Wolfe and J. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture," Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 1991.

22. M. Zhang and K. Asanovic, "Highly−Associative Caches for Low−Power Processors," Kool Chips Workshop, International Symposium on Microarchitecture, December 2000.

23. B. Zheng, J. Tsai, B. Zang, T. Chen, B. Huang, J. Li, Y. Ding, J. Liang, Y. Zhen, P. Yew, C. Zhu, "Designing the Agassiz Compiler for Concurrent Multithreaded Architectures," Workshop on Languages and Compilers for Parallel Computing, August 1999.