Journal of Instruction-Level Parallelism 6 (2004) 1–16 Submitted 02/04; published 04/04

CASH: Revisiting hardware sharing in single-chip parallel processor

Romain Dolbeau DOLBEAU@IRISA.FR
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
André Seznec SEZNEC@IRISA.FR
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France

Abstract

As the increasing of issue width has diminishing returns with superscalar processor, thread parallelism with a single chip is becoming a reality. In the past few years, both SMT (**S**imultaneous **M**ulti**T**hreading) and CMP (**C**hip **M**ulti**P**rocessor) approaches were first investigated by academics and are now implemented by the industry. In some sense, CMP and SMT represent two extreme design points.

In this paper, we propose to explore possible intermediate design points for on-chip thread parallelism in terms of design complexity and hardware sharing. We introduce the CASH parallel processor (for ***C**MP **A**nd **S**MT **H**ybrid*). CASH retains resource sharing a la SMT when such a sharing can be made non-critical for implementation, but resource splitting a la CMP whenever resource sharing leads to a superlinear increase of the implementation hardware complexity. For instance, sparsely used functional units (e.g. dividers), but also branch predictors and instruction and data caches, can be shared among several "processor" cores.

CASH does not exploit the complete dynamic sharing of resources enabled on SMT. But it outperforms a similar CMP on a multiprogrammed workload, as well as on a uniprocess workload.

Our CASH architecture shows that there exists intermediate design points between CMP and SMT.

1. Introduction

Due to both technology improvements and advances in computer architecture, new generation general-purpose processors feature a long execution pipeline (20 stages on the Intel Pentium 4 [1]) and a superscalar execution core. However, as increasing issue width with superscalar processor has diminishing returns, thread parallelism with a single chip is becoming a reality. In the past few years, two approaches for implementing thread level parallelism on a chip have emerged and are now implemented by the industry: CMP (**C**hip **M**ulti**P**rocessor) [2, 3, 4, 5] and SMT (**S**imultaneous **M**ulti**T**hreading) [6, 7, 8, 9].

A CMP design essentially reproduces at chip level the (low-end) shared memory multiprocessor design that was used with previous generation machines. Due to the advance of technology, multiple 2 or 4-way issue (relatively) simple execution cores can be implemented on a single chip (for instance the recently released IBM Power4 processor [5, 10]). The underlying idea advocating for CMPs is that most of the benefits of increasing the issue width will be counterbalanced by a deeper pipeline. It also implies that CPU intensive applications will have to be parallelized or multithreaded.

0. This work was partially supported by an Intel grant.

At the other end of the design spectrum, the SMT paradigm relies on a completely different approach. As pointed out by the designers of the canceled EV8 microprocessor [9], SMTs are essentially designed to achieve uniprocess performance with multithread performance being a bonus. But the SMT processor supports concurrent threads with very low granularity. That is, instructions from the parallel threads are concurrently progressing in the execution core and shares the hardware resources of the processor (functional units, caches, predictors, ...). The main difficulty with SMTs implementation is induced by the implementation of a wide issue superscalar processor; the hardware complexities of the renaming logic, the issue logic, the register file and the bypass network increase super-linearly with the issue-width.

CMP and SMT represent two extreme design points. With a CMP, no execution resource, apart from the L2 cache and the memory interfaces, is shared. A process cannot benefit from any resource of distant processors. On a parallel or concurrent workload, after a context switch, a process may migrate from processor P_i to processor P_j leading to loss of instruction cache, data cache and branch prediction structures warming. On the other hand, with a SMT, single process performance is privileged. Total resource sharing allows to benefit from prefetch effect and cache warming from other threads in a parallel workload.

In this paper, we propose a median course, the CASH parallel processor (for ***C**MP **A**nd **S**MT **H**ybrid*). Instead of an all-or-nothing sharing policy, one can share only some of the resources. CASH retains resource sharing a la SMT when such a sharing can be made non-critical for the implementation, but resource splitting a la CMP whenever resource sharing leads to a superlinear increase of the implementation hardware complexity. For instance, sparsely or rarely used functional units (e.g. dividers), but also branch predictors, instruction caches and data caches, can be shared among several "processor" cores on a single-chip parallel processor. On the other hand, CASH keeps separated the major parts of the execution cores (rename logic, wake-up and issue logic, bypass network, register files) where wider issue implies higher hardware and design complexity.

CASH can not exploit the complete dynamic sharing of resource enabled on SMT (particularly for single process) but retains part of the resource sharing advantage on concurrent workloads and parallel workloads. It is able to use the whole capacity of instruction and data caches and branch predictors with single process workloads. A process migration from processor P_i to processor P_j would not affect the performance of the I-cache and branch prediction structures. With a parallel application a thread prefetches the instructions and warm-up the predictor structures for one another. Sharing the data cache leads to similar prefetching and also avoids coherency traffic induced by the distributed L1 caches in a CMP.

The remainder of the paper is organized as follows. In Section 2, we present in details the rationale for the CASH parallel processors. Our simulation framework is presented in Section 3. Section 4 analyzes simulation results on CASH parallel processors. Finally, Section 5 summarizes this study and presents future directions for this work.

2. Revisiting resource sharing for parallel processors

In few years, thread parallelism on a chip has moved from a concept to a reality. There are two major approaches: Chip MultiProcessor (Figure 1) and Simultaneous MultiThreading (Figure 2). When considering hardware resource sharing among the processes (illustrated in gray on Figures 1 and 2), the two approaches represent two extreme design points.

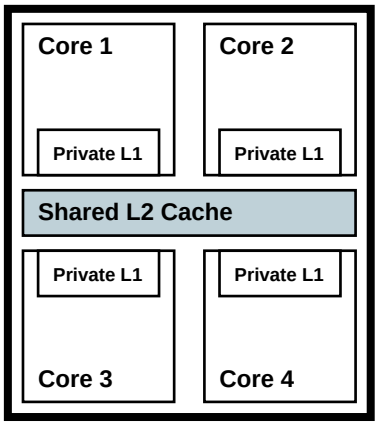

Figure 1: Diagram of the sharing in a single-chip multi-processor.

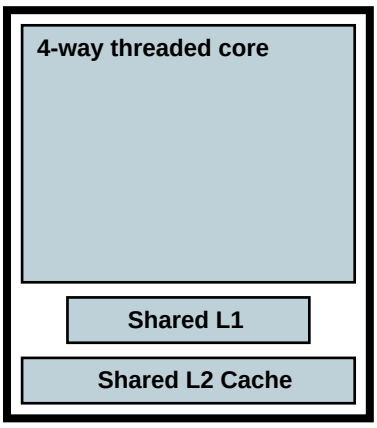

Figure 2: Diagram of the sharing in a simultaneous multithreading processor.

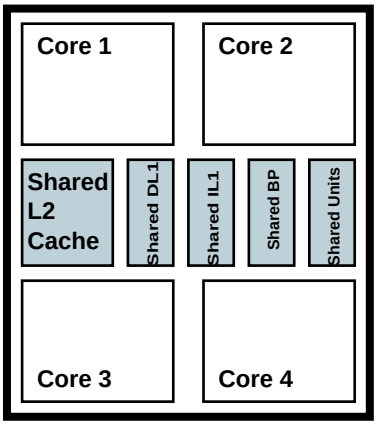

Figure 3: Diagram of the sharing in the proposed CASH.

In this section, we first recall that the complexity of the execution core of a superscalar processor (i.e. the pipeline stages ranging from renaming to retirement) increases superlinearly with the issue width ([11]. For the pipeline stages ranging from register renaming to instruction validation, there is no comparison between the hardware complexity of a 4*N-issue superscalar SMT processor and the complexity of four N-issue processor in a CMP configuration.

On the other hand, resources such as the I-cache, the data cache or the branch prediction structures can be shared at reasonable hardware cost and pipeline overhead. This stands also for long-latency functional units such as integer multipliers and dividers.

Therefore, it becomes natural to consider intermediate designs between CMPs and SMTs with separate execution cores, but with shared instructions and data caches, branch prediction structures, and long-latency functional units. We call such an intermediate design, CASH (for ***C**MP **A**nd **S**MT **H**ybrid*). CASH is illustrated in Figure 3.

Whenever a manufacturer chooses to implement multiple execution cores rather than a very wide issue execution core, CASH represents an alternative to a pure CMP architecture.

2.1 Complexity of wide issue superscalar core

For the sake of simplicity, we assume the following pipeline organization: instruction fetch, instruction decode, register renaming, instruction dispatch, execution, memory access, write back, retire (see Figure 4). Some of these functional stages may span over several pipeline cycles.

In a wide-issue superscalar processor, the hardware complexity of some of these functional stages increases super linearly and even near quadratically with the issue width. This is due to the dependencies that may exist among groups of instructions that are treated in parallel. All the paths must be dimensioned to be able to treat all the in-flight instructions as if they were all possibly dependent.

For instance, let us consider the register renaming stage, the wake-up and issue logic and the operand fetch.

The register renaming stage includes a global dependency check within a group of N instructions to be renamed in parallel. The complexity of this dependency checking is quadratic with the number

Figure 4: The stages of the simulated pipeline.

of instructions. It also requires a single cycle read and update of a register map table. In a single cycle with two operands and one result instructions, 2N entries must be read and N entries must be overwritten.

The complexity of the wake-up and issue logic also increases more than linearly with the issue width. Each entry in the wake-up logic must monitor the producer for each source operand. The number of comparators in each entry in the wake-up logic increases linearly with the issue width. As the total number of the in-flight instructions must also be increased, the total number of comparators in the wake-up logic raises near quadratically. If any instruction can be launched on any functional unit able to execute it (for instance, any ALU operation can be executed on any integer ALU), the complexity of the selection logic raises also quasi-quadratically with the issue width. As an alternative, distributed selection logic can be implemented at the expense of some freedom (and performance). For instance, the functional unit executing the instruction can be preselected at dispatch time.

The complexity of operand fetch comes from two factors: the register file and the bypass network. The complexity of the register file in term of silicon area increases more than quadratically with the issue width. The silicon area required by a single register cell increases with the square of the number of register file ports. At the same time more physical registers are needed when the issue width is increased. The access time of the register file also increases dramatically. The number of bypass points required in the bypass network increases linearly, but the number of entries at each bypass point increases more than linearly and the length of result buses also increases.

2.2 Sharing resources in CASH

The control of dependencies inside a wide-issue SMT superscalar processor results in very high hardware complexity on several functional stages in the pipeline. However some hardware resources needed by every processor could be shared on a multiple execution core processor. For CASH processors, we envisioned sharing long latency units, instruction caches, data caches and the branch predictor.

2.2.1 SPARSELY USED LONG LATENCY UNITS

Some functional units are sparsely used in most applications and feature a long latency, for instance integer multipliers and dividers. These execution units also require a substantial silicon area compared with a simple ALU. Traditional uniprocessors must feature a copy of these functional units. Up to the mid 90's, the performance of these functional units were often sacrificed to tradeoff smaller silicon area against latency with not so often used instructions.

With CASH, several execution cores share long-latency execution units, since the contention for their use induces a low overhead. Control around these units will slightly lengthen their latency. However an extra cycle on a 20-cycle latency `div` is much less a concern than on a 1-cycle latency `add`.

As an alternative, one may consider to invest the total silicon area of the N slow dividers in a CMP in a single more efficient divider in CASH. One may also consider sharing a single pair of floating point units between two execution cores in CASH, or implementing a single multimedia unit.

2.2.2 INSTRUCTION CACHE AND BRANCH PREDICTOR SHARING

Replicating the instruction cache and branch prediction structures for every processor in a multiple execution core is a waste of resources.

Instruction cache To feed a N-issue execution core, the instruction fetch engine must deliver an average throughput equal or higher than N instructions per cycle. The average size of the instruction block that could be fetched in parallel if the front-end is able to bypass not-taken branches is relatively high (8 to 12 instructions on many integer applications, more on floating point applications).

Therefore, four 2-way issue execution cores (or two 4-way) could share a single instruction cache. At each cycle, a single process is granted the access to the instruction cache and a large block of contiguous instructions is read. Processors are granted access to the instruction cache in a round robin mode. Other more aggressive allocation policies can be designed as for SMTs [8].

Even with a capacity four times as large, a shared instruction cache hardware complexity would be in the same range as the hardware complexity of four instruction caches for four 2-way processors.

Branch prediction structures *Accurately* predicting the address of the next instruction block is a challenging problem when the prediction must be performed in a single cycle.

For instance, both Alpha EV6 [12] and Alpha EV8 [13] use (not so accurate) single-cycle line predictors backed with a 2-cycle complex instruction address generators that feature target re-computation, complex conditional branch computation, return address stack (and jump predictor for EV8). Whenever line prediction and address generation disagree, instruction fetch resumes with the address generated by the complex address generator. Due to this hierarchical structure, overfetching is implemented on the Alpha EV8 (i.e. two 8-instructions blocks are fetched on a single cycle.)

In conjunction with the shared instruction cache proposed above, the next instruction address generator in CASH is also shared the same way. If N execution cores are implemented then the address for the next block does not need to be valid before N cycles.

The shared instruction address generator hardware cost will be in the same range as replicating N address generators (about the same number of adders, but a wider multiplexor). The total budget for the branch prediction tables may be invested in a single but larger one. Moreover the hardware devoted to line prediction can be avoided while the need for overfetching associated with the poor line predictor accuracy disappears.

A previous study [14] has shown that for global prediction schemes, a benefit on branch prediction accuracy can be obtained from sharing tables, but imposes a return stack for each thread.

2.2.3 SHARING THE DATA CACHE

In CASH, the data cache is shared between several processes. This leads either to the use of a multiported cache or to the use of a bank-interleaved cache.

The use of a multiported cache in a CASH processor avoids arbitration during accesses to the cache. But implementing a real multiported cache where each processor could grab an access to the same cache block during each cycle leads to a very expensive design. The silicon area devoted to a memory cell grows quadratically with the number of concurrent access ports. Moreover, the access time to a multiported memory array is longer than the access time to an equivalent size single ported one. Therefore, on a multiple execution core, replacing the distributed data caches by a single multiported cache is not cost-effective for access time as well as for total silicon area.

For CASH processors, we propose to use a bank-interleaved structure. The cache can be implemented using single-ported banks. Arbitration is performed on the access to the banks and lengthens the access time. The access time will be further increased by a centralized access, i.e. the latency of the cache will be increased by a cycle or more. On the other hand, the total silicon budget devoted to the L1 data caches in a CMP can be invested in a single larger L1 data cache.

2.2.4 BENEFITS AND DRAWBACKS OF SHARING MEMORY STRUCTURES

There are a few similarities between sharing the data cache and sharing the instruction front end.

Some extra hardware logic is induced by sharing instead of replicating the structures: arbitration + interleaving for the data cache, wider instruction fetch path + wider multiplexors + extra instruction buffers before decode for the instruction front end. Thus, sharing induces some extra cache access time and therefore slightly deepens the pipeline.

Some benefits from this sharing appear quite the same way for both structures. First, one can also invest in a single shared structure the total hardware budget that should have been spent on replicating several copies of the structure. The data cache, the instruction cache, the branch predictor can be implemented with larger capacities in comparison with distributed structures. For each structure, the total capacity is shared.

A second benefit of sharing is the simplification of L2 cache access. For instance, with a 4-way CMP, the L2 cache must arbitrate between requests from the four instruction caches and the four data caches (plus the system bus and the memory bus). Coherency must also be maintained between the L1 data caches. Therefore the total access time to the L2 cache is longer on CMP than on CASH.

Benefits of cache sharing are given below.

1. When a multiprogrammed workload is encountered, the capacity is dynamically shared among the different processes. In Section 4, we will see that this situation is most often beneficial for performance.

2. When the workload is a single process, this situation is clearly beneficial since the process can exploit the whole resource.

3. When a parallel workload is encountered, sharing is also beneficial. Instructions are shared, i.e. less total capacity would be needed, and prefetched by a thread for another one. The branch prediction structures are warmed by the other parallel threads when global history schemes are used. The data cache benefits from sharing and prefetching for read-only data as the instruction cache does. Moreover, on a CMP processor, the data caches of the different processors have to be maintained coherent. On CASH, there is no such need for maintaining data coherency. Thus bandwidth on the L1-L2 cache bus is saved.

4. With a shared memory multiprocessor and therefore with a CMP, the migration of a process from a processor P_i to processor P_j would result in a significant performance loss due to a new cold start with cache memory and branch prediction structures. On a CASH processor, this situation is not an issue. As the processors share the caches, migration of a process incurred by the OS does not lead to cold start penalties on the memory hierarchy.

Memory consistency On a CMP, snooping memory accesses is also used to maintain consistency on the speculative loads and stores internal to each processor cores. On CASH we enforce read after read dependencies on the same memory location inside each processor. That is, whenever two speculative loads on word X occur out of order inside a processor, the second load is squashed and re-issued. This allows to ensure that any (external) write on X seen by the first load will be seen by the second load. This is consistent with the Alpha Shared Memory Model (see chapter 5 section 6 in [15]).

2.3 CASH design spectrum

The total execution bandwidth of the CASH processor presented above is in practice limited by its instruction fetch bandwidth. We have assumed that a single block of contiguous instructions is fetched per cycle. For many applications, the average size of such blocks is not higher than 8 to 12.

However, they are possibilities for sharing resources even if total issue bandwidth is higher. For instance, a larger total issue bandwidth could be obtained from a single instruction cache through 1) use of an interleaved bank structures and either fetching for two distinct processors on a single cycle or fetching two (possibly) non-contiguous fetch blocks (as on Alpha EV8 [13]) or 2) use of trace caches [16].

Different ways of resource sharing can also be considered. For instance, with a four execution cores chip, a pair of adjacent chips could share the instruction front-end and long-latency functional units.

However the purpose of this paper is to illustrate the existence and viability of intermediate design points between CMPs and SMTs. Exploring the whole design spectrum is out of the scope of this paper. Therefore, we will limit our study to CASH processor with total execution bandwidth of eight instructions per cycle, that is either four 2-way execution cores or two 4-way execution cores.

3. Simulations framework and methodology

3.1 Simulated "processor" configurations

We simulated two basic configurations for both CASH and CMP, a two 4-way execution core processors (CASH-2, CMP-2) and a four 2-way execution core processors (CASH-4, CMP-4).

CASH and CMP differ by:

- The first-level instruction and data caches and the buses to the second-level cache;
- The non-pipelined integer unit (used for complex instructions such as "divide");
- The non-pipelined floating-point unit (again used for complex instructions such as "divide" and "square root extraction");
- The branch predictor.

Component	Latency	size	associativity
Level one data cache	3 (CASH) 2 (CMP)	128KB (CASH-4) 64KB (CASH-2) 32 KB per core (CMP)	4 (LRU) 8-way banked
Level one instruction cache	1	64 KB (CASH-4) 32 KB (CASH-2) 16 KB per core (CMP)	2 (LRU)
Level two cache	15 (CASH) 17 (CMP)	2 MB	4 (LRU)
Main memory	150	infinite	infinite

Table 1: Latencies and sizes from the components of the memory hierarchy

The considered memory hierarchy is described in Table 1. The branch predictor is an hybrid skewed predictor 2Bc-gskew [17, 13]. The branch predictor uses four 64 Kbits shared tables and a 27 bits global history per thread in CASH, whereas each of the four predictors in the CMP used four 16 Kbits tables and a 21 bits global history.

The other parameters are common to CASH and CMP. CASH-2 and CMP-2 can issue and retire 4 instructions per cycle in each core. Each core features two simples ALUs, one complex ALU (for the various MUL instructions), two simples FPUs, one branch dispatch unit, a pair of memory access units (can together emit one load and one load or store per cycle), one complex ALU (for the various integer DIV instructions) and one complex FPU (for the FDIV and FSQRT instructions). Those last two are shared by all the cores in the CASH processor.

The four cores configurations could issue and retire 2 instructions per cycle in each core, with the same mix of available execution units in each core than above. Each core can support up to 32 in-flight instructions. The register banks (2 per thread, one for floating-point register, one for integer register) have enough renaming registers to rename all in-flight instructions. To avoid simulating the spill-and-fill system code associated with the SPARC windowed registers [18], an "infinite" number of windows was assumed.

The instruction fetch engine differs between CASH and the reference CMP. The conventional CMP is able to fetch 4 (for the 4 cores configuration) or 8 (for the 2 cores configuration) instructions per cycle, every cycle, for each processor. CASH can fetch up to 16 consecutive instructions, but only once every four cycles for a single processor. The CMP fetch engine is slightly more efficient, as it suffers less from short basic bloc terminated by a taken branch. Note that the instruction fetch engine assumed for CMP is very optimistic. Accurate branch prediction in a single cycle is not implemented on processors like EV6 for instance. Dispatching of the instructions is done out-of-order from a 32-instruction queue.

The simulated pipeline is illustrated on Figure 4. In this figure, the grey areas are again those shared by the processor cores: only a single core can do an Instruction Fetch in a given cycle. The Memory Access is also shared, starting with the level 1 data cache (address computation is still private to a core). All other pipeline stages (Instruction Decode, Register Renaming & Instruction Dispatch, Execution, Instruction Retire) are private. The minimum misprediction penalty (fetch to fetch) of a mispredicted branch is 16 cycles, to account for full pipeline flush and recovery of wrong-path instructions (the simulator fetches but does not simulate wrong-path instructions). Note

that on a misprediction, a CASH core waits for its next instruction fetch slot, resulting in up to 3 extra penalty cycles.

3.2 Simulation environment

Simulation results reported in this paper were obtained using a custom, cycle-accurate, trace-driven simulator. Executions traces were extracted using the CALVIN2 toolset[19], based on the SALTO[20] system, and then fed to the simulator. CALVIN2 is an instrumentation system running on SparcV9 hardware that includes an embedded functional simulator of the SparcV9 instruction set. The embedded simulator allows for fast-forwarding through uninteresting sections of code, such as the initialization phase of a program. This feature allows to simulate only the useful computational part of the code, and thus helps obtaining more significant results.

The simulated execution cores were highly detailed. Three simplifications were made ; first, perfect memory disambiguation ; second, perfect translation look-aside buffer and third, a perfect return addresses stack. On our benchmark set, these three components rarely miss when reasonably sized. All other features were accurately modeled, including contention on the various buses to the different levels of the memory hierarchy.

Prior to execution, caches and predictors were warmed by running 20 millions instructions of each benchmark (interleaved by chunks of 25 thousands instructions) *after* skipping the initialization phase of the application. Sets of benchmarks were run together until the completion of at least 10 millions instructions in each benchmark of the set.

The exception was the simulation of the effect of contexts switches: each thread was then run to 50 millions instructions, to be able to have reasonably long time-slice.

3.3 Benchmark set

Our benchmark set is a subset of the SPEC CPU2000 benchmark suite (10 from the "integer" subset, and 10 from the "floating-point" subset). All benchmarks were compiled using the Sun C and Fortran compilers, targeted at the v8plus sub-architecture, with -xO3 and -fast optimizations for C and Fortran respectively.

With a 4 execution cores configuration, 10 integer 4-thread workloads and 10 floating point 4-thread workloads were considered. Each benchmark appears in 4 workloads. A set of chosen mixed FP/int workloads was also run.

With a 2 execution cores configuration, 10 integer 2-thread workloads and 5 floating point 2-thread workloads were considered.

In addition to running fully multiprogrammed workloads, we also ran every benchmark alone on the four considered configurations. This allows to compare single-thread behavior of CMP and CASH.

3.4 Performance comparison methodology

Fair performance comparison of hardware exploiting thread-level parallelism is a difficult issue (see for instance [21]) ; using average IPC may lead to biased results towards high IPC applications.

In order to get a valid comparison of performances, we chose to compare the execution times process per process and for exactly the same slice of the application (i.e, 10 millions instructions of the thread). That is, for a given thread, statistics were extracted after the first 10 millions instructions

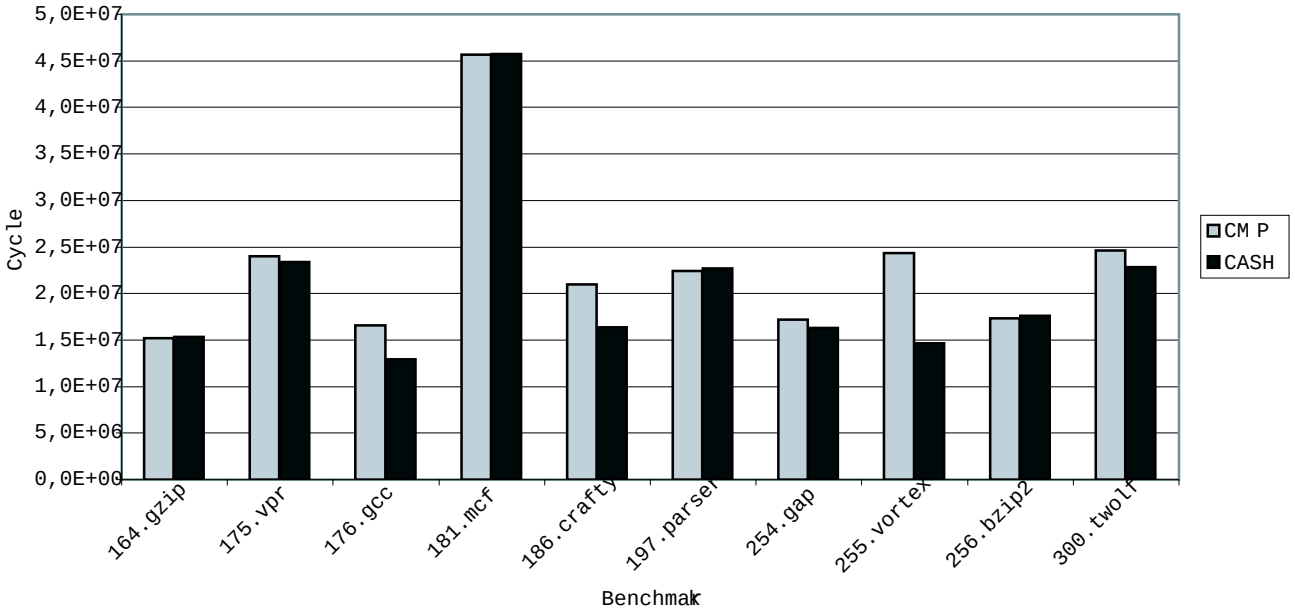

Figure 5: Average execution time of integer benchmarks in groups on the 4 cores configurations

have been executed: the complete group of benchmarks continue to be simulated until the last thread has reached its statistic collection point.

This method allows to compare the behavior of each individual thread on CASH and CMP for a given mix of benchmarks.

4. Simulation results

4.1 Multiprogrammed workloads

Ten fully integer workloads were simulated on the 4-core chips and each benchmark was run 4 times in different environments. No group was performing entirely worse running on CASH than on CMP: at least one of the benchmark in the group performed better (i.e. completed the first 10 millions instructions in a shorter execution time) on CASH than on CMP. Precisely, two groups exhibited degradations on three benchmarks, five groups on only one benchmark, and the remaining three groups showed improvements on all benchmarks.

No single benchmark performed worse on all the four workloads. Three of them (164.gzip, 256.bzip2, 197.parser) performed better on a single run, and by only a tiny margin. One benchmark (181.mcf) did twice better and twice worse. But the remaining six benchmarks always performed better, and sometimes by a wide margin: between 19% and 24.5% for 176.gcc, between 16.9% and 30% for 186.crafty and between 33% and 42% for 255.vortex, an instruction cache-hungry benchmark. On Figure 5, we illustrate the average execution time for the 4 runs for all integer benchmarks. In average, the execution time is improved by 9% when using CASH instead of CMP.

Floating-point workloads exhibit more mitigated behaviors. All of the workloads except one presented mixed results. Three workloads had only one improved benchmark, five exhibited two improved benchmarks, two had three improved benchmarks, and the last one performed better on all

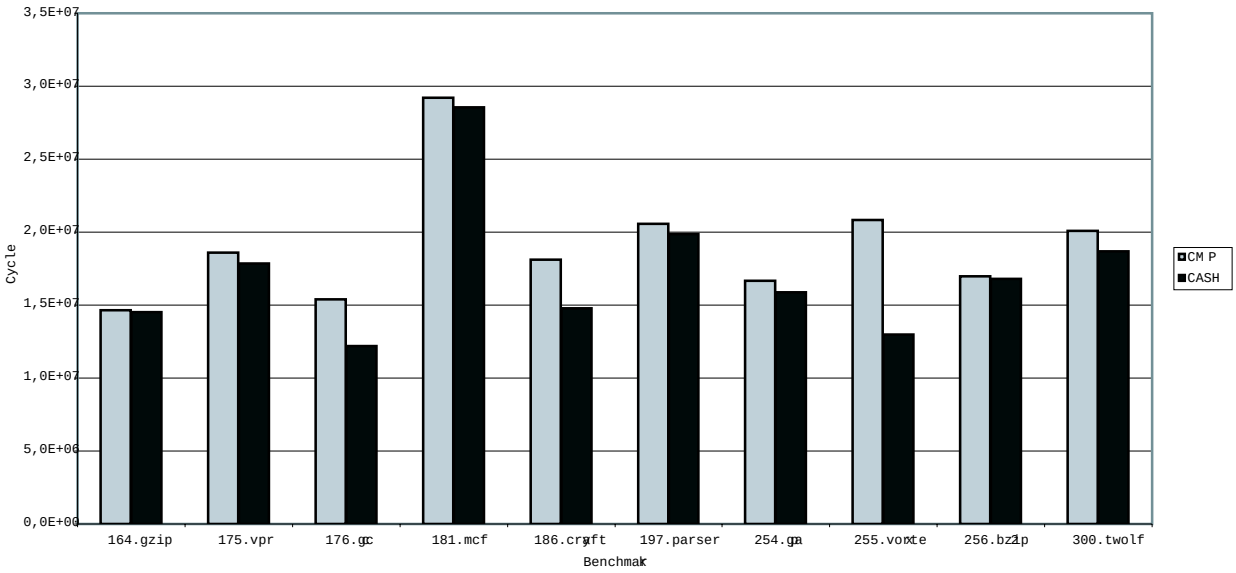

Figure 6: Execution time of integer benchmarks alone on the 4-way chips

benchmarks in the group. No group did worse on all benchmarks. Broken by benchmark, the results are also more mitigated. Four benchmarks (168.wupwise, 171.swim, 191.fma3d and 200.sixtrack) performed consistently better on CASH and four performing consistently worse (172.mgrid, 173.applu, 183.equake and 187.facerec). 178.galgel performed better only once, but with variation always below 1%. Finally 301.apsi improved by about 7% on three runs with one run slowed by about 1%. The average execution time of all floating-point benchmarks improved by only 1.8% going from the regular CMP to CASH.

We also ran selected mixed workloads, with two integer and two floating-point benchmarks. Results were in line with the others, with none of the tested benchmarks performing noticeably worse than usual, and one (191.fma3d) performing better in a mixed workload. This result is due to the larger L1 data cache in CASH.

Looking into specific components of the processor, we notice an almost negligible effect of the shared execution units due to the low frequency of such instructions. Sharing the branch predictor has a relatively small impact with prediction accuracies very similar on both CMP and CASH.

The memory hierarchy on the other hand has a huge impact on performance. As expected, the benchmark with the best results on CASH are those with the most pathological behaviors: 255.vortex benefits from a large instruction cache, 176.gcc has a similar behavior on a more modest scale. When run together, 255.vortex and 176.gcc both exhibit their most modest gain on CASH.

2-core chips 2-core simulations exhibit similar behavior as 4-core simulation, but in a smaller range. A modest average 1.6% gain was encountered for integer workloads on CASH and an insignificant average loss of 0.16% was encountered for floating-point workloads.

4.2 Single process workload

Every benchmark was run alone on all the simulated configurations. On the 4-core chips, the four times larger caches on CASH-4 lead to performance improvement on every single benchmark. These results are illustrated for integer benchmarks on Figure 6.

On the 2-core chips, performance for 2 of the 20 benchmarks (both floating-point) is lower on CASH-2 than on CMP-2, but by no more than 0.6%. All others benchmarks were faster on CASH, albeit for some by a tiny margin.

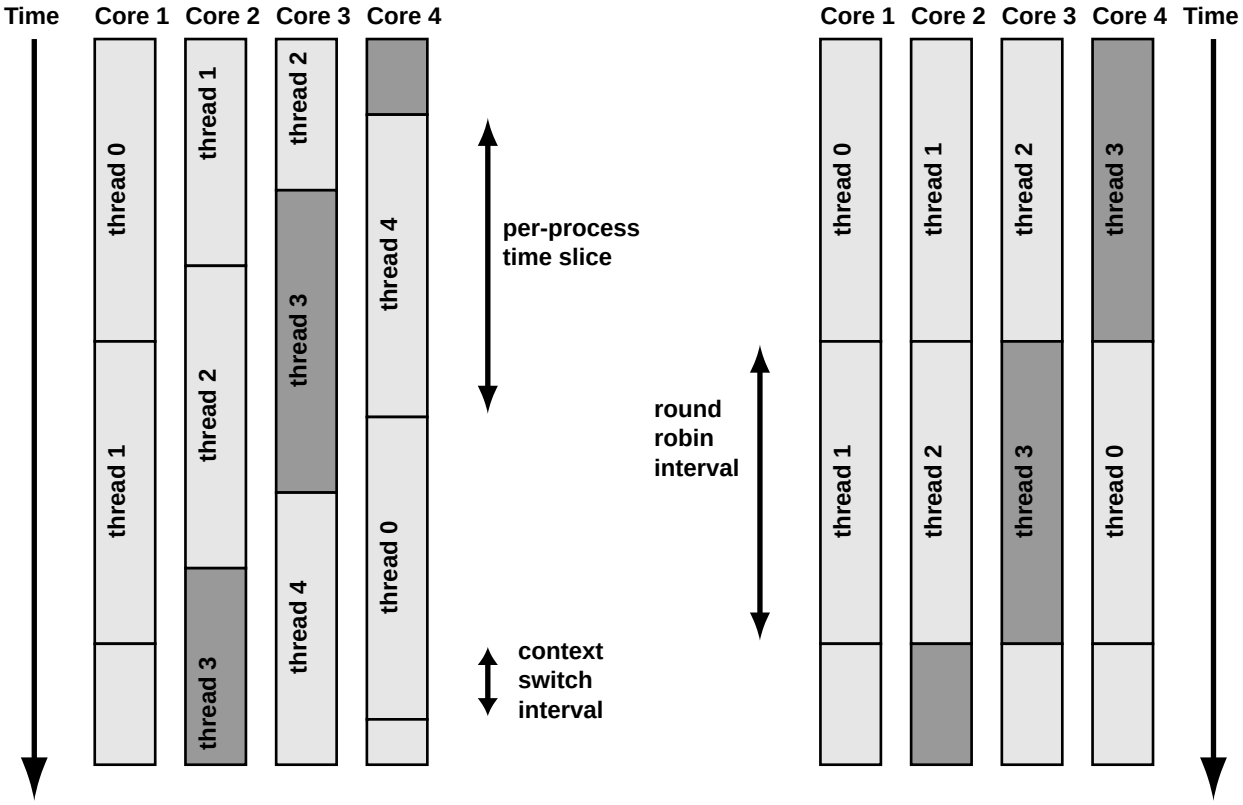

Figure 7: Thread rotation on 4 cores processors. Left is the 5-thread case, right is the 4-thread case.

4.3 Context switching workload

We also studied the respective behaviors of CASH and CMP when the workload includes context switches. Our first such workload was the usual 4-thread workload, but we added context switches in a round robin fashion: at regular interval, all threads would move to the previous core (see right side of figure 7.)

We also simulated an "excessive" workload, i.e. a workload with more running processes than the number of available CPU cores. A time-slice was defined, and each process would run on one core for that time-slice before being suspended. Another process is then made to run on this core. The round-robin effect of running 5 processes on 4 cores is shown in figure 7, left side: each

process will return on a different core as the one it previously ran on, with a switch occurring inside the processor once each (time-slice in cycles / number of cores) cycles. For instance, thread 3 starts on core 4, leave room for thread 4, and return on core 3 after thread 2 is suspended.

Simulations results were consistent with what one might expect: CASH is more resilient, performance-wise to such contexts switches than a CMP. If the number of contexts switches is small in comparison of the number of instructions simulated, then the threads are slightly slowed on both CASH and CMP. There is a distinct advantage for CASH in the first case (four threads round-robin) but this advantage is not distinguishable from the usual sharing effect in the second case (five threads on four cores). The long interval during which the fifth thread is sleeping results in the disappearance of its data and instructions from the caches.

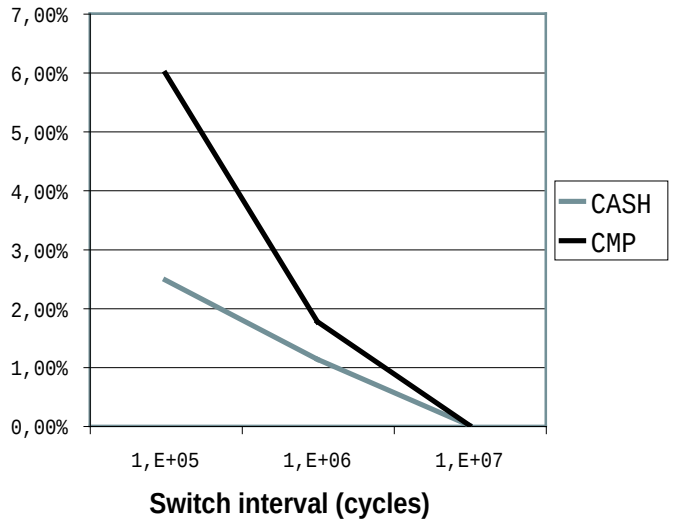

Figure 8: Average performance loss, with a 10 millions cycles interval as reference.

When the number of context switches is raised, the advantage of CASH is also raised (see figure 8): a thread running on a CMP always starts in a "cold" environment with regards to memory and prediction structures, whereas on CASH the environment is still warm (first case) or is not completely cooled (second case). In particular, the shared branch predictor in CASH suffers much less from very frequent contexts switches than the 4 branch predictors in our CMP model.

Even though this effect is not very noticeable for large time-slice, it is nonetheless interesting: It was shown in [22] that the major performance cost of a context switch is due to the cache structures and not the context saving. CASH diminishes this cache-trashing effect, and therefore would allow operating system designer to use shorter time-slice for a finer granularity of process execution.

4.4 Parallel workload

Finally, we also simulated a subset of the SPLASH-2 [23] benchmarks suite, parallelized using POSIX threads. Traces were extracted from a 4-thread execution, after all threads were started (i.e. inside the function used in the PARMACS CREATE macro).

All simulated benchmarks exhibited some level of inter-threads sharing and prefetching. The positive effect of this sharing became more visible as the working set to cache size ratio became higher, i.e., by enlarging the working set or diminishing the cache size. CASH is the best performer whenever this ratio is high, thanks to a higher hit ratio on the memory hierarchy.

When most of the working set (data and instructions) fit in the cache(s), CASH and CMP are close performers, the benchmark execution time becoming dominated by internal execution. For instance, on our traces, the radix kernel from SPLASH-2 was dominated by the floating-point instructions (more than 62% of the instructions, with a large number of double-precision multiplications). If the cache are big enough, the CMP naturally takes the edge thanks to a lower L1 latency. An exception to this is the behavior of the barnes application, were internal execution is dominated by long-latency floating-point instructions (see below subsection 4.6).

4.5 About sharing the L1 data cache

Sharing the L1 data cache in CASH is not automatically positive with multiprocess workloads, since sharing induces an extra latency cycle with loads and may also generate some bank conflicts.

Assuming four execution cores, we ran simulations, both integer and floating-point, on a processor similar to CASH, but with the private, faster access L1 data cache and the longer latency L2 cache similar to CMP. The performance of this version was marginally (in average 1%) better than the performance of CASH. In particular, in most of the cases where CASH-4 was slower than CMP on an individual benchmark, performance close to CMP-4 was obtained.

On the other hand, for single threaded workload, an almost consistent performance advantage was obtained using a shared data cache, with only a handful of benchmarks performing better on the private caches model.

The marginal performance improvement on multiprogrammed performance is low, compared with the benefit of larger caches for incomplete workload, and the benefit of sharing data and instructions in a multithreaded workload.

4.6 About sharing sparsely used functional units

As explained above (see 2.2.1), sparsely used functional units such as dividers can usually be shared between cores without sacrifying performances. Being “sparsely” used, there is usually very little contention on these units. When a single thread make intensive use of for instance the FDIV instruction, it does not suffer from contention if the other processes in the workload do not use the floating-point divider.

However, the situation may be different for a multithreaded workload. When all threads exhibit the same behavior (as is usual for parallelized code), contention may occur. For instance, the barnes application in the SPLASH-2 benchmarks sets exhibits such a behavior. Performance on the CASH processor suffers, as the 4 threads are effectively serialized over the floating-point divider functional unit.

Therefore, it can be cost-effective on CASH to implement a single, bigger, faster unit. This results in slightly better performance for single-threaded code than can use only one divider, and avoids excessive penalty on multithreaded code. In our experiment, halving the latency of the floating-point divider was enough to make CASH more than a match for CMP on the barnes benchmark, even though CASH was nearly 20% slower with the usual full-latency unit on this specific multithreaded benchmark.

We also made experiments with CASH using shared floating-point units, even for more frequent instructions. Instead of connecting two units to each of the four cores, we only used four units, each connected to two adjacent cores. Each core had access to two units, but no two cores were connected to the same two units. The unit responsible for the floating-point division and square root instructions shared by all cores, with the usual high latencies. We then ran our floating-point benchmarks sets on this configuration.

As one might expect, on all floating-point workloads, all benchmarks performed worse running on this configuration while mixed workload exhibited a similar behavior as regular CASH. The benchmark with the higher proportion of floating-point instructions in the trace (173.applu, 172.mgrid, 171.swim) naturally suffered the most, especially when put together on adjacent cores. Other benchmarks were slowed by a smaller amount.

5. Conclusion and future work

The combined impacts of the advance in silicon technology and the diminishing return from implementing wider issue superscalar processor has made hardware thread parallelism a reality in commodity processors. Both CMPs and SMT processors are now offered by manufacturers.

In this paper, we have shown that there exists an intermediate design point between CMP and SMT, the CASH parallel processor (for *CMP And SMT Hybrid*). Instead of an all-or-nothing sharing policy, CASH implements separate execution cores (as on CMPs) but shares the memory structures (caches, predictors) and rarely used functional units (as on SMTs). Whenever sharing allows to use larger memory structures (for caches and branch predictors), or to save material (rarely used functional units, but also line predictor), CASH implements it, even if it induces longer pipeline access and some arbitration. Whenever sharing induces superlinear complexity increase, CASH implements separate hardware: for instance register files, wake up and selection logic, and the bypass network.

The simulation results presented in this paper illustrate that CASH competes favorably with a CMP solution on most workloads. In future studies, we will explore a wider design space. We also want to study (mostly) software solutions to enhance single process workload performance. For instance, sharing the caches among the core allows an helper thread (running on a different core) to prefetch instructions and data for the main thread.

References

[1] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, Q1 2001.

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang, "The case for a single-chip multiprocessor," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 2–11, 1996.

[3] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, pp. 79–85, Sept. 1997.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing," in *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture: June 12–14, 2000, Vancouver, British Columbia*, (New York, NY, USA), ACM Press, 2000.

[5] K. Diefendorff, "Power4 focuses on memory bandwidth: IBM confronts IA-64, says ISA not important," *Microprocessor Report*, vol. 13, Oct. 1999.

[6] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the* 22^{nd} *Annual International Symposium on Computer Architecture*, pp. 392–403, ACM Press, June 22–24 1995.

[7] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, pp. 322–354, Aug. 1997.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice : Instruction fetch and issue on an implementable simultaneous MultiThreading processor," in *Proceedings of the* 23^{rd} *Annual International Symposium on Computer Architecure*, (New York), pp. 191–202, ACM Press, May 22–24 1996.

[9] K. Diefendorff, "Compaq chooses SMT for alpha," *Microprocessor Report*, vol. 13, Dec. 1999.

[10] J. Petrovick, "POWER4 chip integration," in *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), IEEE Computer Society Press, 2000.

[11] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in 24^{th} *Annual International Symposium on Computer Architecture*, pp. 206–218, 1997.

[12] R. E. Kessler, "The Alpha 21264 microprocessor: Out-of-order execution at 600 MHz," in *Hot chips 10: conference record: August 16–18, 1998, Memorial Auditorium, Stanford University, Palo Alto, California* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), IEEE Computer Society Press, 1998.

[13] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeidès, "Design tradeoffs for the ev8 branch predictor," in *Proceedings of the* 29^{th} *Annual International Symposium on Computer Architecture: May 25–29, 2002, Anchorage, Alaska*, (New York, NY, USA), ACM Press, 2002.

[14] S. Hily and A. Seznec, "Branch prediction and simultaneous multithreading," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, (Boston, Massachusetts), pp. 169–173, IEEE Computer Society Press, Oct. 20–23, 1996.

[15] R. L. Sites, *Alpha Architecture Reference Manual*. Digital Press and Prentice-Hall, 1992.

[16] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the* 29^{th} *Annual International Symposium on Microarchitecture*, (Paris, France), pp. 24–34, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 2–4, 1996.

[17] A. Seznec and P. Michaud, "De-aliased hybrid branch predictors," Technical Report RR-3618, Inria, Institut National de Recherche en Informatique et en Automatique, 1999.

[18] D. L. Weaver and T. Germond, eds., *The SPARC Architecture Manual, version 9*. PTR Prenttice Hall, 1994.

[19] T. Lafage and A. Seznec, "Combining light static code annotation and instruction-set emulation for flexible and efficient on-the-fly simulation," Technical Report PI-1285, IRISA, University of Rennes 1, 35042 Rennes, France, Dec. 1999.

[20] E. Rohou, F. Bodin, and A. Seznec, "SALTO: System for assembly-language transformation and optimization," in *Proceedings of the Sixth Workshop Compilers for Parallel Computers*, Dec. 1996.

[21] Y. Sazeides and T. Juan, "How to compare the performance of two smt microarchitectures," in *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[22] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California), pp. 75–85, 1991.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. of the* 22^{nd} *Annual International Symposium on Computer Architecture*, (Santa Margherita, Italy), pp. 24–36, June 1995. Published as Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95), ACM SIGARCH Computer Architecture News, volume 23, number 6.