Performance/Area Efficiency in Chip Multiprocessors with Micro-caches

Michela Becchi
Washington University
Computer Science and Engineering
One Brookings Drive
St. Louis, MO 63130-4899
+1-314-935-4306
mbecchi@cse.wustl.edu

Mark Franklin
Washington University
Computer Science and Engineering
One Brookings Drive
St. Louis, MO 63130-4899
+1-314-935-6107
jbf@cse.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
One Brookings Drive
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@cse.wustl.edu

ABSTRACT

This paper proposes the use of very small instruction caches, called micro-caches (μ-caches), consisting of tens to hundreds of bytes, at the bottom of the instruction delivery hierarchy in chip-multiprocessors (CMP). Multi-core architectures place a novel emphasis on the performance/area efficiency of processor cores, and we note that traditional instruction cache sizes reflect an emphasis on hit-rate performance rather than efficiency. In brief, μ-caches reduce the area footprint of individual cores, thus allowing additional cores to fit within a given die area. We use commercial design tools and a commercial processor core to evaluate this tradeoff in the context of high-performance networking, where CMP architectures have had their greatest commercial impact to date. Our results suggest that the use of u-caches can yield a 25% improvement in efficiency relative to traditional hierarchies. In our evaluation, we consider a range of architectural options (cluster organization, non-blocking caches, cache parameters) and justify our conclusions while accounting for the errors inherent in die area estimates.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Multiple-instruction-stream, multiple-data-stream processors (MIMD)*.

General Terms: design, performance.

Keywords: Chip multiprocessor, networking workload, cache hierarchies.

1. INTRODUCTION

For well-documented reasons of power efficiency and performance, high-performance processor designs increasingly consist of homogeneous clusters of relatively simple processors. Particularly in throughput-oriented applications (e.g., networking and communications) where program structure is amenable to a multi-processor implementation, overall system performance goals such as area and power efficiency are best met with dense CMP solutions. Intel's 16-core IXP 2800 network processor [19] and Cisco's Silicon Packet Processor [12] are two high-profile examples of such clustered CMP systems.

CMP architecture designs generally emphasize area efficiency of the replicated processor cores; in particular the performance/area efficiency of the cache hierarchy. In this paper, we explore the use of very small instruction caches, called μ-caches, which range in size from 64 to 256 bytes. In a CMP system with L1 instruction μ-caches, processors are arranged in clusters that share an on-chip L2 instruction cache. This shared cache is configured much like a traditional L1 instruction cache. Provided that the use of μ-caches does not reduce performance greatly, a substantial area savings can be achieved due to the reduced number of L1 I-caches. If, for example, the traditional I-cache accounts for a third of total core area — the other components being the processor core proper and the data cache (D-cache) — then, in the best case, effectively removing the I-cache from each core will allow a cluster of 4 cores with μ-caches to occupy roughly the same area as a traditional 3-core cluster. Note that in commercial processor cores used in networking (such as Tensilica Xtensa), the instruction cache occupies 30% of the total core area. However, even in the Intel IXP 2800 network processor, which uses a single-ported RAM rather than a cache to store instructions at each processor core, instruction storage accounts for 20% of die area consumed by a processor core[1].

This paper's main contribution is in the proposed use of μ-caches for instruction delivery in CMP processors. To evaluate the effectiveness of this proposal, we present a simulation-based experimental study that compares the performance/area efficiency of μ-caches to that of traditional instruction cache hierarchies. Our study has two important characteristics. First, rather than trying to demonstrate that μ-caches are effective in the general case, we restrict our study

CF'07, May 7–9, 2007, Ischia, Italy.

[1] Based on discussions with Intel architects.

to applications that are commonly or easily deployed on clustered CMP processors and consist of pipelines of networking and communication processing kernels. These kernels are used to construct workloads that correspond to three approaches to mapping programs to multiple cores. Second, we obtain performance and area estimates by the use of a commercial design environment.

Namely, we have built our system using Tensilica's Xtensa design tools [7]. This environment allows development of cycle-accurate system simulations with multiple processor cores, as well as sophisticated on- and off-chip memory components and interconnects. The Xtensa environment is best known for its configurable and user-extendable processor instruction set architecture (ISA), although we do not exploit that capability in this study. Notably, the Xtensa environment was used to design Cisco's Metro NP.

The remainder of the paper is organized as follows. Section 2 introduces the architecture and describes our experimental methodology and setup. Sections 3, 4 and 5 report and analyze the results on three different program deployment scenarios. Section 6 relates our work to the state of the art. Finally, Section 7 summarizes and concludes the paper.

2. ARCHITECTURE AND METHODOLOGY

In this section we describe our system model, simulation infrastructure, and benchmarks used for evaluation.

2.1 System model

Figure 1 shows a traditional design and our proposed design. In the traditional design, processors and caches are grouped within clusters with separate caches associated with each processor. In our design, processors in a cluster each have their own private µ-cache, µ$i, and share an instruction cache, I$. Thus, the µ-cache effectively forms the first level of instruction cache hierarchy. The data and the instruction caches, D$i and I$, are connected to off-chip data and instruction memories.

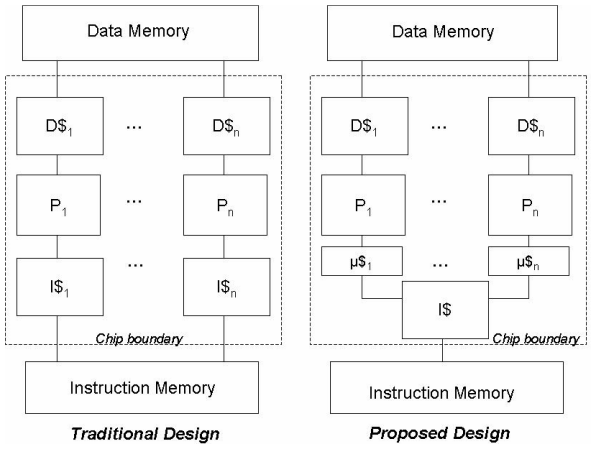

Figure 1: Logical model of a cluster: traditional vs. µ-cache based design.

The parameters of the system are listed in Table 1, where the cells containing parameter ranges are highlighted in bold type. In our evaluation, this organization is compared to a traditional one where each processor has a private 4 KB L1 instruction cache having a one-clock cycle hit latency. In our proposed architecture the µ-caches also have a one-clock cycle hit latency, however, on a miss, access to the shared instruction cache takes 3 additional clock cycles. We admit that this setting is optimistic in case of large cluster sizes, where routing distance can imply higher hit rates. On the other hand, our simulations do not neglect the penalty in access time due to concurrent requests to the shared cache. The shared instruction cache has been sized to ensure at least a 99% hit rate for each of our benchmarks. The memory latency has been chosen assuming a clock rate of 300 MHz, corresponding to Tensilica Xtensa design tool estimate.

The shared I-caches have been modeled as single-port devices. We limit our investigation to single-port devices since the resulting performance is acceptable in most cases, and additional ports require additional area models which are not readily available. Thus, if multiple requests are received at a shared I-cache in the same clock cycle, all but one will be rejected. Each rejected request will be repeated one clock cycle later. Shared instruction caches may support hit-under-miss (i.e., are non-blocking) through a configurable number of miss status holding registers (MSHRs). Later, we analyze the effect of cache contention on various cluster sizes and MSHRs count.

We restrict our study to mono-threaded processors, addressing the reader to [22] for issues concerning instruction cache design in a multi-threaded scenario. Notice that while some network processors, such as Intel's IXP, use multithreading, others, such as Cisco's Silicon Packet Processor which is also based on the Xtensa processor core, do not. Therefore, our study maintains a practical significance despite this assumption.

2.2 Simulation Infrastructure

Our simulation environment is built with the Tensilica Xtensa design tool. Among other features, the tool provides a cycle accurate system simulator allowing software emulation of systems consisting of an arbitrary number of interconnected components, such as processors, memories, and interconnecting devices. In particular, configurable and extensible processor cores are provided, allowing for L1 I and D caches (1K to 32K), local (on-chip) and system (off-chip) memories, on-chip device-to-device connectors and hardware-supported lock objects. Moreover, custom devices can be defined according to a given API. The tool provides

Table 1: System parameters

$\mu\$_i$	***Size***	***256B/128B/64B***
	Associativity	***DM/2-way***
	Cache line	***16B/8B***
	Hit latency	1 clock cycle
I$	*Size*	4KB
	Associativity	DM
	Cache line	16B
	# MSHRs	***1/2/4/8/16***
	Hit latency	3 clock cycles
D$	***Size***	***16KB/8KB***
	Associativity	DM
	Cache line	64B
	Hit latency	1 clock cycle
Off-chip memories	*Latency*	50 clock cycles
Cluster	***n (# of cores)***	***1/2/4/8/16***

area estimates for cores, caches and memories; in our evaluation a 0.13 μm LV process is assumed.

The core used in our simulation is the 5-stage Xtensa LX Microprocessor [8], a 32-bit RISC, scalar processor. For our purposes, the Xtensa cache model provided is deficient in three respects. First, its caches can be no smaller than 1KB, and much smaller μ-caches are of interest. Second, the default cache model cannot be arranged in a hierarchy. Third, the model does not allow the cache to be shared by more than one device. Consequently, we implemented custom devices to simulate μ-caches, hierarchies, and shared caches. The functional and temporal correctness of our components has been verified both by validating correct program behavior, and by crosschecking miss rates, latencies and global execution times when using those components in place of standard caches.

In the Xtensa environment, external devices and cores are interconnected through a number of different port types, some generic (e.g., PIF: "processor interface ports") and others intended to connect to local instruction or data memories. We found that the PIF ports cannot sustain required instruction fetch request rates, thus we have our μ-caches attach to processors via local ports.

The use of local ports has two implementation implications that are important when using the Xtensa design environment. First, it constrains the address space cacheable in our custom devices to 256 KB. This is acceptable given the limited code size of our benchmarks. Second, local ports do not tolerate delays: they always expect a memory response in the same clock cycle as the corresponding memory request. To circumvent this, when a core misses in its cache, we stall that core within the simulator until the miss is serviced and ready on the local port. This differs from the normal circumstance in which the simulated core control logic stalls itself until the miss is serviced. This modifies the simulator implementation but does not change execution time or program behavior.

Table 2: Benchmark applications

Name	Application
VITERBI	Probabilistic pattern search algorithm
DRR	Deficit round robin queue maintenance
FRAG	IP header fragmentation: checksum
CAST	Encryption
REED	Reed-Solomon FEC encoding
ZIP	Data Compression
CRC	Checksum computation
MD5	Message Digest algorithm
URL	URL-based switching

2.3 Benchmarks

Broadly, our benchmarks are program kernels drawn principally from communications applications. The networking and communication programs are selected from two well-known suites of benchmarks: CommBench [5] and Netbench [6]. The communications programs have been selected to represent typical applications for both traditional routers (header processing applications-HPA) and programmable routers (payload processing applications-PPA). An added application, viterbi, implements a probabilistic pattern search algorithm based on dynamic programming. All programs have been compiled through the proprietary Xtensa compiler xt-xcc, which is a customization of gcc for the Xtensa Tensilica platform. Optimization level 3 has been used in all cases.

Table 2 summarizes the benchmarks (code and input datasets can be downloaded from [11], [12] and [13]). Programs frag, cast, reed and zip belong to Commbench, crc, md5 and url to Netbench, and drr to both. Table 3 reports a measure of the instruction working set size obtained by profiling the applications and determining the size of the most frequently fetched portions of code. Data was collected by separately running each program and tracing all instruction fetches.

Note that, for all benchmarks but cast, gzip, md5 and url, a few dozen 8-byte words account for 98% of all fetches. The programs cast and md5 have the worst behavior. In particular, cast trace shows about 30% coverage through 15 to 19 distinct fetches and a sudden change to 98% coverage with 193 requests (motivated by the presence of a 170 instruction loop). In case of md5, 32 distinct fetches allow 20% coverage, and a sudden change to 75% coverage is observed with 560 distinct fetches. Finally, gzip and url exhibit an intermediate behavior: as it can be observed from Table 3, their coverage increase happens more gradually. These results

Table 3: Number of distinct 8-byte instruction fetches accounting for given percentage of the total instruction fetches

	viterbi	*drr*	*frag*	*cast*	*reed*	*gzip*	*crc*	*md5*	*url*
60%	9	8	7	193	4	11	16	560	21
70%	10	8	11	193	4	11	16	560	39
80%	12	10	22	193	5	47	16	754	93
90%	14	14	26	193	6	91	17	754	151
98%	15	19	36	193	6	154	23	880	267

suggest that for most of the benchmark programs there exists a high degree of locality, and thus the use of very small caches may not lower overall performance.

2.4 Mapping Programs to Cores

Consider next the deployment of benchmark programs to the available processor cores. As described earlier, dense clustered CMP systems are often used to implement processing pipelines in which an application consists of a series of processing steps. In this study, we will examine the three standard application deployment approaches.

First, we note that each of the above applications has a main loop with the following structure: at every iteration a different piece of data is read from memory and processed. The data processed represents a packet header for HPAs, a packet payload for PPAs, and a protein motif in the case of Viterbi. Thus, each program serially processes jobs from a work queue. In our CMP context, we may have multiple programs consuming work from the same queue. To accommodate this, our systems use locks to ensure safe access to this shared queue.

There are three natural ways to deploy programs on a clustered CMP system. This is shown in Figure 2, which considers the case of four processor cores:

Case A: All the cores within a cluster execute the same task (i.e., run the same program).

Case B: All the cores within a cluster execute a different task.

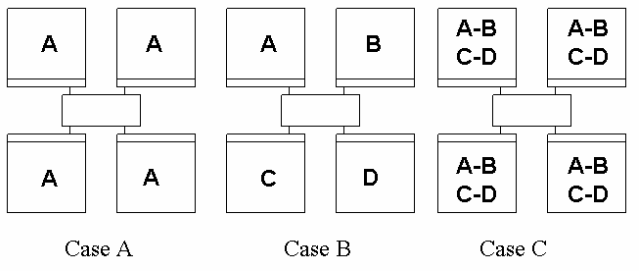

Figure 2: Three classes of application structure shown with a 4-core, 4-program example.

This represents a deployment scenario often used in the networking context (e.g.: IXP network processors).

Case C: All the cores within a cluster execute the same task sequence (i.e., a run-to-completion model). The overall program consists of a main loop: at each iteration, one basic iteration of each program is performed. Programs executing within the sequence are not synchronized between cores. This represents a deployment scenario where the developer is freed from the task of breaking the applications into multiple parallel stages.

Notice that case A represents the scenario which most directly motivates the introduction of μ-caches and cache sharing. We can envision adopting such a deployment in order to take advantage of the packet level parallelism characterizing networking applications. Case B cannot take advantage of any code sharing and requires that multiple programs be stored in the shared instruction cache; however, an overall area savings may still occur. Case C is again characterized by code sharing, but with an increased code size.

3. CASE A: All the cores within a cluster execute the same task

In this section we evaluate the use of μ-caches when all the cores in a cluster run the same program. In our processor configurations, we use an 8KB data cache. Similar computational performance results have been observed using a 16KB data cache. We first analyze system behavior for the different parameter choices in Table 1. We then carry out a performance/area analysis with the goal of finding the optimal configuration in terms of processing power per area.

3.1 Effect of μ-cache size

Consider first the base case where only a single processor core is present along with a single μ-cache, I-cache and D-cache. This is introduced for comparison purposes since, in a real implementation, there would be no need for the μ-cache in a single core system.

Figure 3 indicates the effectiveness of using the proposed instruction delivery hierarchy. A comparison is made between the base case, and the traditional case where no μ-cache is present and instruction fetches access a single I-cache directly. The graph shows, for each program and for different μ-cache sizes, the ratio between the execution time resulting from using a μ-cache and the traditional instruction cache without one. For the base case, the μ-cache is backed by an instruction cache with a 3 cycle hit time, whereas the traditional case has a directly attached cache with a single cycle hit time. A ratio of 1 indicates that the μ-cache does not introduce a penalty. As can be seen, most configurations are within 20% of 1. This benefit is exploited in clusters with

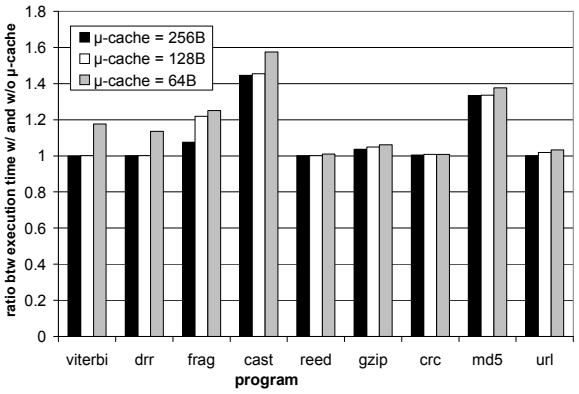

Figure 3: Ratio between execution time using µ-cache (base case) vs. connecting directly to the instruction cache in a traditional configuration (one core).

shared L2 instruction caches. In brief, the area saved by replacing L1 caches with small µ-caches and a single shared L2 cache can be used to either increase performance for a given area, or decrease area for a given level of performance.

Figure 4 reports the hit rate in the L1 µ-caches base case. These results support and explain the previous results. As observed, and as could be predicted by the analysis of Table 3, all the programs except for cast and md5 exhibit good behavior. Moreover, gzip and url perform better than the observation of Table 3 would have suggested: this may be due to the fact that the groups of instructions which account for the greater percentage of the total fetches are loaded into cache in successive program phases, such that the miss rate within each phase remains moderate. That is, Table 3 does not consider temporal locality characteristics of gzip and url. This consideration also explains the fact that md5 performs better than cast and experiences the same miss rate despite its bigger memory footprints.

We note that most of the programs (frag being an exception) do not experience a significant performance loss when the size of the µ-cache is reduced from 256B to 128B, whereas

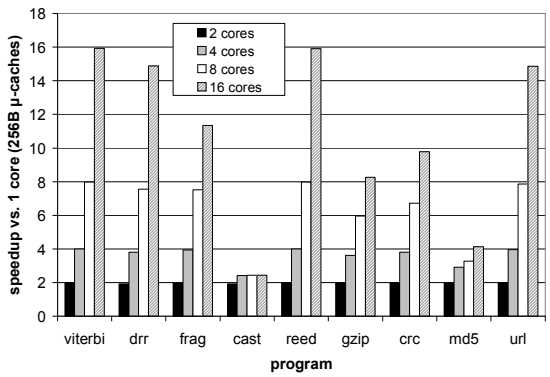

Figure 5: Speedup of different size clusters over 1 core (256B µ-cache with one MSHR).

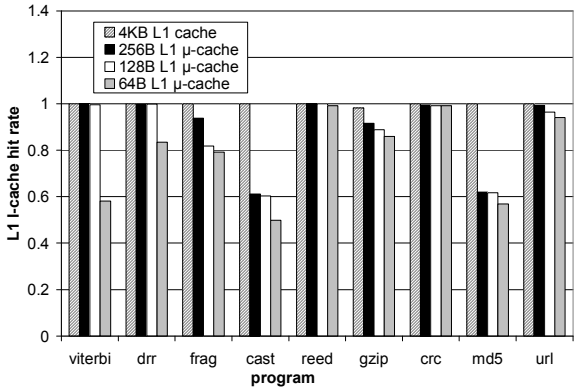

Figure 4: Hit rate of L1 I-cache (one core).

most of them are sensitive to an additional halving of the µ-cache size to 64B.

3.2 Effect of µ-cache associativity

The previous results are based on direct mapped µ-caches. Additionally, two-way set associative µ-caches have been tested for the three cache sizes listed above. The results observed are very similar, and in most cases slightly worse, than the ones of direct mapped µ-caches. This fact confirms the intuition that additional complexity is not necessary when using caches that hold few instructions, since those instructions tend to belong to simple loops that reside in adjacent memory words. Moreover, we note that set associative caches occupy more die area than direct mapped ones (because of the additional logic needed to handle different ways). Consequently, only direct mapped µ-caches have been considered in the rest of this work

3.3 Effect of µ-cache line size

The previous results are based on a 16B cache line in both the µ-cache and instruction cache. For our benchmarks, this line size is best in the traditional design where only a L1 instruction cache is used. Because of the limited size of the µ-

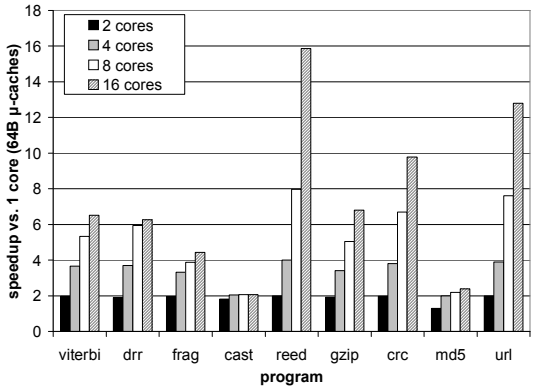

Figure 6: Speedup of different size clusters over 1 core (64B µ-cache with one MSHR).

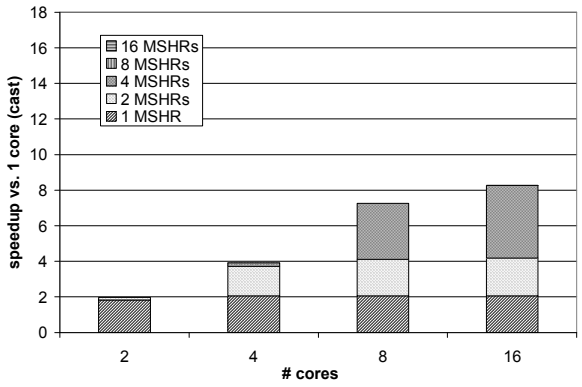

Figure 7: Cluster speedup vs. 1 core (cast & 64B µ -cache).

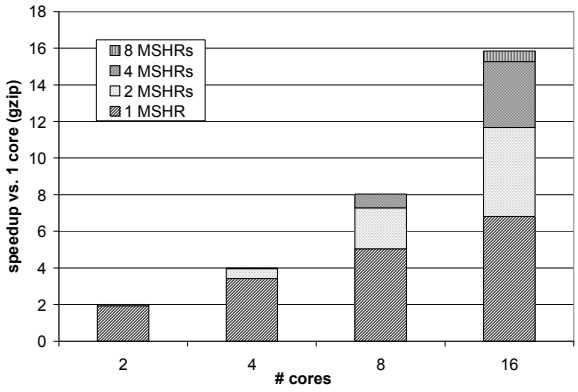

Figure 8: Cluster speedup vs. 1 core (gzip & 64B µ-cache).

caches, we have considered using a smaller cache line; in particular, 8B cache lines have been evaluated with all three cache sizes listed above. From the analysis of the instruction reference traces, it has been observed that the Xtensa processor typically fetches 8B quantities. Therefore, smaller cache lines do not make sense.

Simulation results show a performance decrease when halving the µ-cache block size from 16B to 8B. A relatively small loss is seen for all benchmarks except for a 25% performance decrease in the case of cast and md5. This behavior can be explained as follows. As mentioned, it is likely that small µ-caches will store instructions belonging to loops and are adjacent in memory. Reducing the cache line may therefore increase the number of compulsory misses. For programs having a frequently used working set fitting the µ-cache, those misses are limited in number, and thus may cause only a small performance loss. However, in case of programs exceeding the size of the cache, frequent reloads from lower level caches and the resulting loss of performance will result from the use of small cache lines. Thus, for the remainder of this paper only 16B blocks for both µ-caches and instruction caches are considered.

3.4 Effect of cluster size

We have tested configurations with 2, 4, 8 and 16 cores per cluster. Using an even numbers of processors generally permits symmetric on chip arrangements with homogeneous core-to-shared cache distances. Additionally, µ-cache sizes of 256B, 128B and 64B have been considered. Figures 5 and 6 show the speedup over the base single core configuration with 256B and 64B µ-caches and one MSHR for the shared cache. We observe that programs with high µ-cache hit rates (viterbi, reed and, to a less degree, drr and url) have a linear speedup proportional to the number of cores in the cluster.

Too many simultaneous accesses from processors to the shared cache cause contention, which decreases the speedup. This effect, observed with gzip, crc and frag, increases with cluster size. In the case of cast and md5, the under-sizing of the µ-cache causes frequent use of the shared cache: cache contention heavily limits the speedup beginning with a 4-core cluster.

As the size of the µ-cache is decreased to 64B (Figure 6), the effect of cache contention grows for nearly all the benchmarks. Reed, which uses a very restricted number of distinct instruction fetches (see Table 3), is an exception and still has optimal behavior across all the cluster sizes. Cast and md5 see an additional loss in performance, but their behavior does not substantially deviate from that observed in the 256B case. Gzip, crc and url experience only a modest performance decrease. For the rest of the benchmarks, the effect of increasing shared cache contention can be observed starting with a 4-core cluster.

3.5 Non-blocking shared caches

Two aspects of cache contention reduce performance as seen above: i) contention due to the use of a single ported shared cache, and ii) contention due to blocking subsequent requests on a miss (i.e., use of a blocking cache). As mentioned earlier, we do not consider multi-ported caches, however, we do evaluate non-blocking caches. This is done by varying the number of miss status holding registers (MSHRs) which varies the number of misses that can be sustained while servicing subsequent hits. Using a single MSHR (as in Figure 5 and 6) allows the cache to handle only one request at a time. Note that since our cores are scalar and generate only one instruction memory request at a time, the maximum number of MSHRs simulated is equal to the cluster size. Figures 7 and 8 show the results of varying the number of MSHRs for cast and gzip. As in Figures 5 and 6, the speedup over the base single core configuration with a 64B µ-cache (worst case) is reported. Note that cast is the program which, due to its relatively large working set size, showed poor performance for any cluster size greater than two; conversely, gzip performance scaled with the number of cores.

Table 4: Area occupancy of the basic components

Component	Area occupancy
core	0.61 mm^2
4 KB, DM cache	0.42 mm^2
4KB, 2-way cache	0.69 mm^2
8 KB, DM cache	0.54 mm^2
µ-caches	42,000 μm^2+137 μm^2/ 32-bit word

In the case of gzip, we see that scaling MSHRs with cluster size achieves near ideal speedup (proportional to the cluster size). As a matter of fact, while not shown in the graph, four MSHRs are sufficient for all cluster sizes. In this case, we note that adding further logic to make the shared cache multi-ported would not be of any help. All the benchmarks except cast and reed exhibit a behavior similar to gzip.

In the case of cast, adding MSHRs is also beneficial up to a cluster size of eight cores. As the cluster size increases, contention due to the use of a single ported cache plays a significant role in reducing the overall speedup. We note that, also in this case, four MSHRs are sufficient to achieve the maximum benefit of supporting hit-under-miss.

3.6 Performance/area analysis

In this section different configurations are evaluated on an area-equivalent basis with the configuration yielding the best performance/area ratio being the most suitable for deployment in a system with either one or multiple clusters. That is, the analysis which follows can be seen from two perspectives: the most efficient configuration will i) increase performance for a given area by providing the opportunity for adding cores or ii) decrease the area and thus the cost associated with achieving a given level of performance. Notice that in the evaluation that follows clusters of different sizes are also compared to a traditional design which does not make use of µ-caches.

Our area computation is limited to cores, caches and µ-caches. In particular, in the case of cores and traditional caches, the area estimates provided by the Tensilica Xtensa tools have been used. For µ-caches, a formula provided by Tensilica has been utilized. Table 4 displays this data assuming a 0.13 µm LV process. The µ-cache formula (last row) includes a constant factor (an estimate of the control logic) and a variable factor (dependency on µ-cache size). When applying this formula, both data and tag arrays have been taken into consideration. Note that the cache area estimates provided by the Xtensa tool is based on single-ported, blocking caches and do not account for the area consumed by MSHRs and their control logic. This fact, coupled the inevitable uncertainty of any area estimates, motivate our parametric area analysis in the next section.

Figures 9 and 10 report the performance/area ratio (MIPS/mm2) when 256B µ-caches are used. With the data provided in Table 4, smaller µ-caches do not reduce area enough to justify their use. A 316MHz clock frequency, the estimate obtained from the Xtensa tool, has been assumed.

Figure 9 reports the results for a blocking shared cache. In this scenario, all programs but cast and md5 see a benefit when µ-caches are introduced. In the case of viterbi, drr, reed and url, performance increases with cluster size. However, because of the increasing shared cache contention, the incremental performance benefit is reduced as the number of cores grows. In case of frag, gzip, and crc , cache contention causes the performance increase to stop at a several cluster sizes (mostly four).

With non-blocking shared caches (Figure 10), the behavior of viterbi, drr, reed and url does not substantially change (they were already optimal in the previous case). The performance/area for Frag and gzip however, increases up to a cluster size of sixteen. Finally, while cast and md5 benefit from a non-blocking shared cache, we note that a 256B µ-cache is not efficient for those programs. Thus, the use of µ-

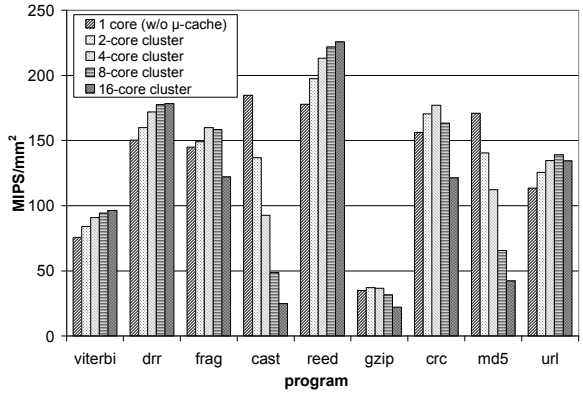

Figure 9: Performance/area analysis using 256B µ-caches and no MSHR on shared cache.

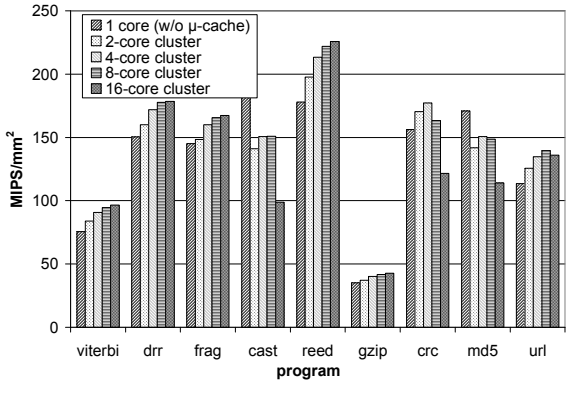

Figure 10: Performance/area analysis using 256B µ-caches and the number of MSHRs equaling the cluster size.

caches yields a MIPS/area improvement for all the programs except cast and md5. Defining performance improvement, PI, as follows:

$$PI = \frac{\text{MIPS/mm}^2_{\text{cluster}} - \text{MIPS/mm}^2_{\text{uniprocessor}}}{\text{MIPS/mm}^2_{\text{uniprocessor}}}$$

where "uniprocessor" corresponds to the traditional design without a µ-cache, then using the data in Table 4, the maximum theoretical PI is 0.37 . This is a theoretical maximum since it assumes: 1) µ-caches that consume no area, and 2) µ-caches that have no slowdown relative to a traditional cache. Results using our more realistic configuration values show that the maximum performance improvement in PI (for a cluster size of 16) ranges from 15% (frag) to 28% (viterbi). The use of a non-blocking shared cache allows cluster sizes up to sixteen to be beneficial. However, contention for the single port limits the benefits of larger clusters.

3.7 Accounting for uncertain area estimates

As mentioned, the results presented are based on area estimates. We have noted that these estimates do not take into consideration additional control logic needed for non-blocking operation in the shared instruction cache. More importantly, however, is the fact that the area required to implement a given logic function can vary widely and cannot be known with certainly until layout and routing. For this reason, we attempt to generalize our results in two ways. First, we report performance improvement PI as we vary the size of the µ-cache as a fraction of the traditional I-cache it replaces. This first comparison assumes that the I-cache consumes about 27% of the total core area (including the core proper and the D-cache), as shown in Table 4. This assumption leads to the second generalization in which we vary the fraction of the total core consumed by the I-cache, and quantify PI .

Figure 11 reports PI as we vary the area of the µ-cache relative to the I-cache it replaces in 8 and 16-core clusters for two representative programs, viterbi and frag (best and worst cases). These results assume that that I-cache area accounts for 27% of total core area (Table 4). In previous results, the 256B µ-cache represents 8% of the area of the I-cache. As expected, as the µ-cache fraction increases, the speedup decreases. However, the µ-cache provides an increase in efficiency up to very large fractions. For frag, for example, the crossover point falls at 55%, at which time the use of µ-caches ceases to improve performance/area efficiency. Thus, even if our relative area estimates are off by a factor of 6x, µ-caches will still improve efficiency.

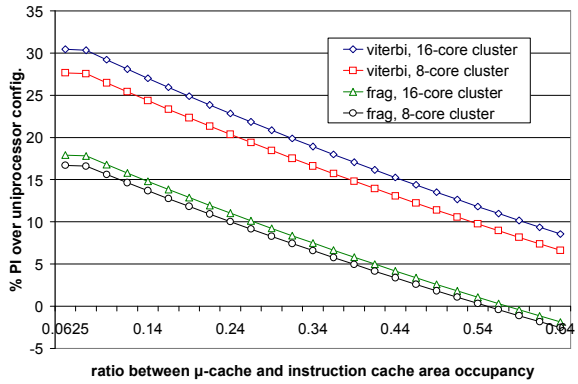

Figure 11: Dependence of performance improvement *PI* on µ-cache area occupancy.

Figure 12 reports PI as we vary the area of the I-cache relative to the entire core (including the core proper and the D-cache). Note that the estimate data in Table 4 implies that the I-cache consumes 26% of the total area. As intuition suggests, an increase of the relative I-cache area increases the benefit of replacing it with a µ-cache. The figure assumes a conservative 15% ratio between µ-cache area and I-cache area. As can be seen, speedup increases exponentially with relative I-cache area.

4. CASE B: Cores execute different tasks

In this section we consider the scenario in which each core in a cluster executes a different program. In the simulations performed for this scenario, since each program has a different execution time we set the number of iterations, or assigned jobs, differently for each program so that all programs see nearly the same total execution time on a given configuration. This ensures that all cores within a cluster will be active all the time.

The benchmarks used in this phase are: drr, frag, reed and cast. In the cases where the number of cores exceeds the number of available distinct programs, we emulate the presence of more programs by executing all instances in distinct address spaces. Thus, the programs appear to be distinct to the caches, albeit with similar working set sizes and access patterns. The µ-caches are direct mapped, since, as explained in Section 3.2, this is the best configuration when each program is run separately.

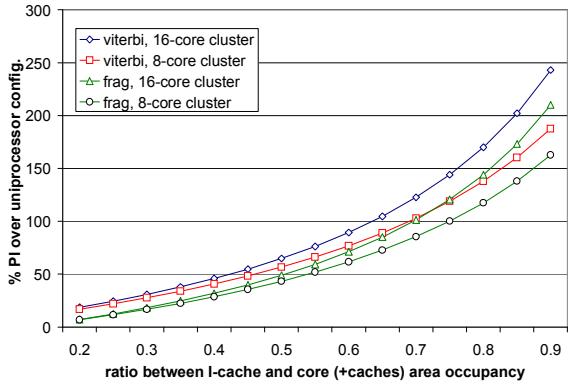

Figure 12: Dependence of performance improvement PI on I-cache area occupancy (assuming that µ-cache area is 15% of the I-cache).

Figure 13 shows the results on dual core configurations with 256B µ-caches. All possible combinations of the aforementioned programs have been taken into consideration; as can be seen, there is a relatively large variation in performance variability across the programs. Observe that the overall behavior is negatively affected by the program with the worst performance. As shown earlier, µ-caches do not support cast efficiently; consequently, all configurations with cast show poor performance. Otherwise, however, we see that program combinations without cast experience improved efficiency.

In Figure 14 we compare 2-, 4- and 8-core clusters; in the dual-core case, the results reported in Figure 13 have been averaged. In both the 4- and 8- core cases all four listed programs are executed; in the latter case, two distinct instances of each are deployed on distinct cores as described previously. Note that we do not increase the shared cache size beyond 4KB despite the increased number of programs.

As can be seen, despite the presence of cast, a 4-core cluster still achieves a small performance benefit (or no loss) over a multiprocessor configuration using private I-caches. On the other hand, further increasing the cluster size causes a drop in performance. For larger numbers of programs, the shared cache is inadequate. The same simulations were performed using a 2-way shared instruction cache. The (contained) performance benefit was compensated by the bigger area occupancy, resulting in an overall PI efficiency loss for all except the 8-core configurations.

In conclusion, the use of µ-caches is not generally effective when executing a different programs on each core in a cluster. For small clusters, with 4 or fewer cores, small gains or losses in efficiency are seen. 8-core clusters, however, experience large reductions in efficiency.

5. CASE C: Each core executes all tasks

In this section we consider the third scenario in which each core executes the same sequence of different tasks. In the context of networking applications, it is generally the case that for each incoming packet a sequence of tasks is performed (e.g., fragmentation, encryption, redundancy encoding, data compression, etc.). Thus, in a run-to-completion approach, packets from a given flow are assigned to separate processors and all the tasks required are executed on each of the processors. This provides for packet level parallel processing and high packet throughput and is similar to some of the commercial network processors available.

Overall program execution on each core has a main loop that iterates over incoming packets and performs the required sequence of tasks on each packet. To explore this approach, two groups of simulations were performed: the first uses the 3-task sequence frag-reed-cast, and the second uses the 4-task sequence frag-reed-cast-gzip. The program drr has not been considered in this phase since it operates in parallel on groups of packet headers spending a very negligible amount of time on a single packet.

The performance/area results, assuming non-blocking shared caches, are reported in Figure 15. We make the following observations. First, the performance improvement increases with the cluster size, even if the relative gain gets lower for a greater number of cores. This is analogous to what was seen in case A and can be motivated similarly.

Second, the performance improvement varies from 8% to 25%. In general, performance is dominated by the longest executing task: reed in the first case and zip in the second one. The presence of cast, whose performance is limited by the use of µ-caches, slightly slows down the overall performance. However, since the execution time of this task is lower than the one of the others, the use of µ-caches is still effective.

In general, it can be observed that, in these scenarios, the fact that the 4KB instruction cache must support a greater code base (i.e., the sum of the single tasks) does not result in a bottleneck. Each task can be considered as being a single phase within the whole program execution and µ-caches provide enough support for each of these phases. The mandatory misses in the µ-caches occurring at phase changes (that is, at task transition) constitute just a small fraction of the overall memory accesses. Thus, the use of the instruction cache and the I-cache contention is limited.

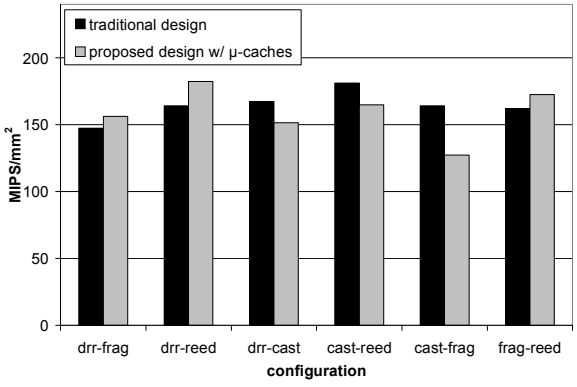

Figure 13: Case B - performance/area with dual-core configuration and different program combinations.

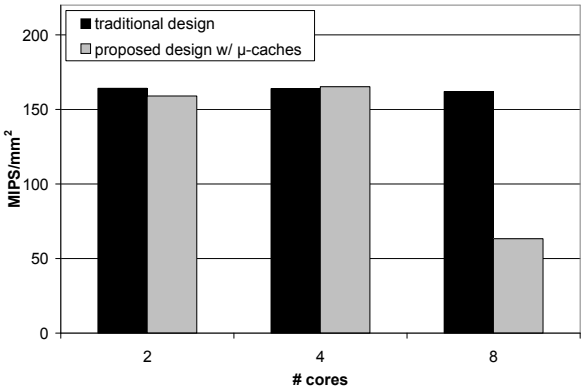

Figure 14: Case B - performance/area with different cluster sizes (256B µ-caches).

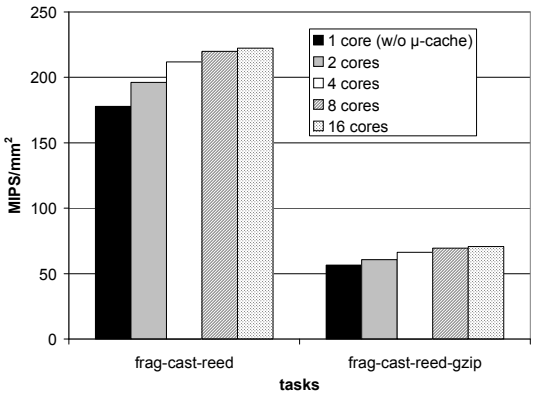

Figure 15: Case C - Performance Improvement vs. uni processor configuration (256B µ-caches)

6. RELATED WORK

In the past few years the design of effective CMP cache organizations has been addressed in the literature. Several papers have focused on techniques to improve access latency to the shared L2 cache in a two level cache hierarchy. [15], [16], and [17] have extended non-uniform cache access techniques (NUCA and NuRAPID) to CMPs. The basic idea is to divide large caches into banks having different access latencies associated with the requesting processors. Data is then mapped to the banks (with the tags in the NuRAPID case) to reduce access time. These techniques, however, are advantageous only if big caches are assumed (often the case for data caches). Our work differs in that we focus on instruction caches and on workloads having high hit rates (>99%) with small I-caches (<4KB). Such caches are too small to justify attempts at utilizing the more complex techniques described above.

The use of caches of restricted size has also been proposed in [14]. However, such work differs from ours in several ways. First, its analysis is restricted to a single processor, and does not include considerations about cache sharing and clustering. Second, the benchmarks considered are different: while we focus on networking workloads, the authors of [14] based their study on media applications. Third, the performance metric that they aim at optimizing is power-delay, while we focus on performance/area.

In [3] an analytical model to predict inter-thread contentions in chip multiprocessor architectures is presented. However, the model focuses on data rather than instructions and is validated on superscalar cores and SPEC benchmarks. In [19] the effect of cache pollution when multiple threads contend for the use of a shared cache is investigated. Sharing the instruction cache among multiple cores executing the same program (as in our study) can be seen as a way to exploit cache pollution.

In [10] the design space for cores and caches in CMPs is explored using an experimental methodology, as we do in this work. However, the focus is on commercial applications and on using multithreading to hide memory latencies. The cache sizes are varied only within standard values (with a minimum of 8KB).

A study focusing on networking applications is proposed in [4], where an analytical model for designing and evaluating architectures for network processors is presented. As in this paper, the authors of [4] aim at minimizing a performance/area efficiency metric; however, there are several crucial differences. [4] presents an analytical model while our study is experimental and based on simulation. Moreover, [4] considers multi-threaded processors and just private L1 instruction and data caches, while we evaluate single-threaded processors, µ-caches and instruction cache sharing. However, it should be possible to extend the analytical model presented in [4] to our architecture.

The combination of considering a compound performance metric (MIPS/area), the use of a µ-cache and a small shared on-chip L1 cache, and the focus on certain streaming applications taken from networking/communications domains differentiates this work from related efforts in this area.

Finally, the term "microcache" was used in [24] in a different context: that work proposed a new cache architecture achieved by giving the compiler control of the cache and by allowing regions of the cache to be allocated to specific program objects.

7. CONCUSIONS

Many highly parallel applications, especially networking applications, have small instruction working sets. Consequently, traditional instruction caches are over-provisioned for these workloads. Moreover, the optimal use of available chip area is a central issue in the design of CMP systems. In this work we have considered trading cache area for processing power by replacing standard-sized I-caches with small caches (µ-caches). These are attached to a shared L2 I-cache whose size and configuration is typical of a traditional L1 I-cache. Cores sharing the same I-cache form clusters.

In order to evaluate the efficiency of the proposed scheme, for collections of relevant programs drawn from the networking/communications domain, we have implemented both traditional and µ-cache-based clusters in the Tensilica Xtensa design environment. We have examined three ways of mapping programs onto processor clusters: in the first, all the cores execute the same task; in the second, each processor executes a different task; in the third, all the cores execute the same sequence of distinct tasks (i.e., the run-to-completion model). Our results indicate that the use of µ-caches coupled with a small shared, non-blocking I-cache improves performance (MIPS/area) for the first and third cases, and has

acceptable performance for the second case (a less likely application scenario).

We have obtained a number of tangible results that are directly applicable to designs associated primarily with the networking environment and pointed the way for designers to examine these tradeoffs with other application benchmarks. Our analysis showed that, for benchmarks run in isolation, a 16-core cluster with 256B μ-caches has on average 22% greater performance/area efficiency than a traditional cluster with 4KB I-caches. Moreover, for an aggregate application consisting of a sequence of programs, the improvement is 25%. Thus, a cluster with μ-caches can provide 25% greater performance in the same amount of area - quite a surprising result and important to designers of real systems.

To generalize the results, and reduce uncertainty inherent in chip area estimates, we provide an analysis that parameterizes the area consumed by instruction caches and shows that μ-caches are effective even with very conservative area estimates. According to Tensilica's area estimates, for instance, each μ-cache takes 8% the area of a 4KB I-cache. However, we showed that, even in the case of the worst program, the μ-cache based design results are beneficial for μ-caches occupying up to 55% the area of an I-cache. Thus, even with rough μ-cache area estimates, the performance gains associated with its use are substantial.

Our evaluation is restricted to simple single-threaded processors (such as the ones present on Tensilica Xtensa chip). The use of μ-caches in multithreaded scenarios is an interesting topic for future work. Also, we plan to apply the idea of improving performance/area through clustering and cache sharing to data caches. However, the micro-cache sizes required will likely be larger since typical data access patterns have larger working sets. Additionally we plan to evaluate μ-caches in other application domains. Finally, the results for Case C suggest that larger programs with episodic execution behavior may also be amenable to μ-cache-based instruction cache hierarchies.

ACKNOWLEDGEMENTS

This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794.

REFERENCES

[1] N.P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, in Proc. of ISCA 17, 1990.

[2] Hennessey and D. Patterson. Computer Architecture a Quantitative Approach. Morgan Kauffmann, Inc., 3rd Ed, 2003.

[3] D. Chandra, F. Guo, S. Kim, and Y Solihin, “Predicting Inter-Thread Cache Contention on a Chip Multiprocessor Architecture”, in Proc. of HPCA-11, 2005.

[4] T. Wolf and M. Franklin, “Performance Models for Network Processor Design,” IEEE Trans. On Parallel & Distributed Systems, V17, N6, 548-561, June 2006.

[5] T. Wolf, and M. Franklin, “CommBench – A telecommunication benchmark for network processors”, in Proceedings of ISPASS, 2000.

[6] G. Memik et alt. , “Netbench: A Benchmarking Suite for Network Processors”, in Proc. of ICCAD 2001

[7] http://www.tensilica.com

[8] Xtensa LX Microprocessor – Data Book, Tensilica, Inc.

[9] Xtensa Instruction Set Simulator-User Guide, Tensilica, Inc.

[10] J.D. Davis, J. Laudon, K. Olukotun, “Maximizing CMP Throughput with Mediocre Cores”, in Proc. of PAC 2005

[11] http://www.ecs.umass.edu/ece/wolf/nsl/software/cb

[12] http://cares.icsl.ucla.edu/NetBench

[13] http://viterbi.wustl.edu

[14] J. Kin, M. Gupta, et alt., “The Filter Cache: An Energy Efficient Memory Structure”, in IEEE Micro 1997.

[15] B. M. Beckmann and D. A. Wood, “Managing Wire Delay in Large Chip-Multiprocessor Caches”, in Proc. of MICRO, 2004.

[16] J. Huh et alt., “A NUCA Substrate for Flexible CMP Cache Sharing”, in Proc. of ICS, June 2005.

[17] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, “Optimizing Replication, Communication, and Capacity Allocation in CMPs”, in Proc. of ISCA, 2005.

[18] R. Kumar, N. P Jouppi and D. M. Tullsen, “Conjoined-core Chip Multiprocessing”, in Proc. of MICRO, December 2004.

[19] A. Agrawal, “Performance tradeoffs in multithreaded processors”, in IEEE Transaction on Parallel and Distributed Systems, 3(5):525-539, September 1992.

[20] M. Adiletta et al., “The Next Generation of Intel IXP Network Processors“, in Intel Tech. Journal, Vol. 6, Iss 3, 2002.

[21] Cisco Systems. Silicon Packet Processor in the CRS-1 Router. http://www.cisco.com/en/US/products/ps5763/index.html

[22] P. Crowley, “Supporting Mixed Real-Time Workloads in Multithreaded Processors with Segmented Instruction Caches.”, in Proc. of the HPCA-10 Workshop on Network Processors and Applications, pages 1-13. Madrid, Spain. February, 2004.

[23] J. Torrellas, C. Xia, and R. L. Daigle, “Optimizing the Instruction Cache Performance of the Operating System”, in IEEE Transactions on Computers, Vol. 47, N. 12, December 1998.

[24] D. May, D. Page, J. Irwin and H. L. Muller, “Microcaches”, in Proc. 6th International Conference On High Performance Computing, pages 21--27, Springer-Verlag, December 1999