# Dynamically Configurable Shared CMP Helper Engines for Improved Performance

Anahita Shayesteh[†], Glenn Reinman[†], Norman Jouppi[£],Suleyman Sair[§], Tim Sherwood[‡]
[†] Computer Science Department, University of California, Los Angeles
[‡] Computer Science Department, University of California, Santa Barbara
[§] Department of Electrical and Computer Engineering, North Carolina State University
[£] HP Labs, Palo Alto

**Abstract**

*Technology scaling trends have forced designers to consider alternatives to deeply pipelining aggressive cores with large amounts of performance accelerating hardware. One alternative is a small, simple core that can be augmented with latency tolerant helper engines. As the demands placed on the processor core varies between applications, and even between phases of an application, the benefit seen from any set of helper engines will vary tremendously. If there is a single core, these auxiliary structures can be turned on and off dynamically to tune the energy/performance of the machine to the needs of the running application.*

*As more of the processor is broken down into helper engines, and as we add more and more cores onto a single chip which can potentially share helpers, the decisions that are made about these structures become increasingly important. In this paper we describe the need for methods that effectively manage these helper engines. Our counter-based approach can dynamically turn off 3 helpers on average, while staying within 2% of the performance when running with all helpers. In a multicore environment, our intelligent and flexible sharing of helper engines, provides an average 24% speedup over static sharing in conjoined cores. Furthermore we show benefit from constructively sharing helper engines among multiple cores running the same application.*

## 1 Introduction and Motivation

While the unceasing march of Moore's law has given computer architects a continually increasing number of transistors to design with, emerging technology trends including poor wire latency scaling, increased power density, and reduced transistor reliability threaten to limit the usefulness of those designs. One approach to providing aggressive cycle times in the face of increasing latency is to deeply pipeline all aspects of the processor, from the branch predictor to the instruction wakeup logic to the cache memories. While deep pipelining has been effective at increasing operating frequency, one of the reasons that performance lags behind that of the trends set by frequency is the increased latency to critical processor loops [2] from deeply pipelined structures. Furthermore, there is extra latency from routing complexity due to large structures [19]. Larger structures also contribute to the growing power density and thermal problems facing modern processor design through greater power dissipation and longer clock wires [3]. To battle these problems, modern machines are increasingly built as a loosely-coupled association of parts that inter-operate in a robust and latency tolerant manner.

At the highest level, the software may be involved to describe the loose coupling between various parts of the computation. Simultaneous Multi-Threading and Chip Multiprocessing based designs [22, 6] have seen widespread adoption both in the academic and industrial environments because, in part, the parallelism they benefit from can be exploited at the architectural level. Since the workload is already explicitly portioned into parts which communicate infrequently, the work may be distributed about the die with only minor effects from the latency of the communication channel between them. However, even at the micro-architectural level the idea of decoupled execution has taken hold. Many researchers have proposed to take the large memory-laden structures that are crucial to single thread performance, and factor them out of the core into specialized *helper engines* [7, 8, 13, 1]. Some structures can be broken in two, with a smaller structure that can hold enough state to provide low-latency for requests that are frequently occurring and a larger, second level, structure that has a capacity sufficient to keep performance high. Other structures can be completely factored from the processor core and have inherent latency tolerance that allows them to avoid becoming critical loops in the processor. Processors are quickly becoming distributed systems on a die.

In such an environment, where we have multiple levels of caches, decoupled branch predictors, separate prefetching and value prediction units, and two level register files, how does one manage all of these resources in a way that will provide adequate performance yet not be wasteful of power and chip area? The problem is further exacerbated by proposals to have multiple threads or multiple cores sharing the same resources [11]. Which resources should be shared, how should they be allocated, and how can we efficiently manage their power? To answer these questions at run-time, we need a set of shared helper engine management policies that can adaptively allocate resources in a way that takes into consideration

the needs of each executing workload.

In this paper, we present novel shared helper engine management policies that base their decisions on a set of carefully chosen observations. We show that through the collection of a few simple to gather run-time metrics, processor resources can be allocated to a set of running programs in a way that is near optimal. We begin by examining a single threaded processor with a variety of helper engines. Our metrics can predict which resources will be most valuable to an executing program so that un-helpful helpers may be put into a low power state yet the overall performance remains almost unaffected. By gathering a statistic as simple as the number of cache hits, we can pick the best overall configuration in a single try. Furthermore, we show how these techniques can be extended to help guide the sharing of resources between multiple cores on a chip. We observe how different helpers have different sharing requirements and demands, and that prior work may not be able to find the best sharing combination. We have implemented a variety of different helper engines and we have extended our simulator to support multiple executing processes so that a detailed analytic treatment of the subject can be presented.

This paper makes the following contributions:

- Analysis of program behavior and resource requirements in an architecture with many major components decoupled via helper engines.
- Proposal of an intelligent mechanism to dynamically tune helper engines to the specific needs of the application for optimal power/performance.
- Extension of this mechanism to intelligent sharing of resources among multiple cores. Prior work has only considered whether or not simple sharing is possible – we demonstrate how different types of helper engines may be shared differently and how sharing must be flexible to attain maximal performance.
- Investigation of constructive sharing when the same application is allowed to share fetch state across different phases of execution and different input sets.

The rest of this paper is organized as follows. In Section 2, prior work on helper engines and CMP sharing is discussed. Simulation methodology can be found in Section 3. Section 4 begins with a description of the different helper engines that we implemented and an analysis of the varying requirements for different applications for different helper engines. We explore a mechanism to enable/disable helper engines, and further extend this to sharing helper engines in a multicore environment in Section 5. We conclude in Section 6.

## 2 Related Work

In [17], Smith proposes a processor implementation that consists of several distributed functional units, each fairly simple and with a very high frequency clock. These units communicate via point-to-point interconnections that have short transmission delays. He then describes how surrounding this simple core pipeline with *helper engines* that perform speculative tasks off the critical path results in enhanced overall performance. Since the helper engines are off the critical path, they can use slower transistors to reduce static power consumption. This is also the motivation behind our factored design, where the speculative structures are reduced to a bare minimum size to support nearby ILP but they are duplicated in larger sizes outside of the critical path for extracting distant ILP. On a follow-up paper [7], Kim and Smith discuss the microarchitecture and ISA that implements this distributed processing paradigm, which utilizes hierarchical register files and a global register file to hold global state. In Section 4, we detail the prior work for each individual helper engine when discussing the implementation of that helper engine.

Sharing some processor resources among cores in a CMP setting was first proposed by Dolbeau and Seznec [5]. Kumar et al. [11] also examine this idea and present a more thorough evaluation of sharing. Dolbeau and Seznec [5] propose the CASH architecture as an intermediate design point between CMP and SMT architectures. A typical CASH architecture shares caches, branch predictors, and division functional units between dynamically-scheduled cores. Kumar et al. [11] also propose resource sharing between adjacent cores of a chip multiprocessor to reduce die area with minimal impact on performance. They investigate the possible sharing of floating-point units, crossbar ports, instruction caches, and data caches and provide detailed analysis of area savings that each kind of sharing entails. Both [5] and [11] examine a round-robin based access model where a resource is allocated to a particular core every cycle. Kumar et al. also investigate a more sophisticated scheme for caches. After suffering a cache miss, a core relinquishes the control of the cache to the other core until the miss is serviced.

Our work differs from these techniques in two dimensions: 1) In addition to reactive (or demand-based) resource sharing such as when sharing caches, we also consider sharing always active resources such as prefetchers and value predictors that run ahead of the execution stream and, 2) We investigate an intelligent approach to assigning resources to cores based on utilization.

## 3 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [4], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Latency values for the caches and register files were obtained using CACTI [16] for a *70nm* process technology at 4 GHz.

| | Core | Helper Engines |
|---|---|---|
| Inst Window | 256 entry ROB | |
| Physical RF | 256 entry register file | |
| BBTB | 128-entry 4-way SA | 1024-entry 4-way SA |
| L1 Data Cache | 8KB 4-way SA, dual port<br>32B block size, 2 cycle lat | 64KB 4-way SA, single port<br>32B block size, 4 cycle lat |
| L1 Instruction Cache | 4KB 4-way SA, single port<br>32B block size, 2 cycle lat | 64KB 4-way SA, single port<br>32B block size, 4 cycle lat |
| Value Predictor | none | 2K-entry stride<br>8K-entry L2 markov |
| Address Predictor | none | 2K-entry stride<br>4K-entry markov |
| Stream Buffer | none | 32-entry FA buffer |
| Branch Mispred | 15 cycles | |
| Core Width | 4-way issue, 4-way decode, 4-way commit | |
| Memory and L2 Cache | 152 cycle memory lat, 2MB, 4-way SA unified cache with a 64B block size and 20 cycle lat | |

*Table 1: Simulation parameters for a single core architecture. These parameters remain the same for the multicore case except for the L2 cache which becomes an 4MB, 4-way set-associative, 4-bank cache shared among all cores.*

We used the SPEC2000 benchmark set for our experiments. Although the results are gathered for all the benchmarks, we only show results for a randomly selected subset of 11 programs in the suite to conserve space in this paper. Details for all benchmarks will be available as a technical report (citation removed for blind review process). The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system with full compiler optimization (`-O4 -ifo`). We picked the 4 most dominant phases as determined by the hardware phase detection technique described in [15] and simulated these phases as representative samples of the program. On average, they accounted for approximately 70% of the execution time of each benchmark. All benchmarks were simulated using the *ref* inputs.

Table 1 presents the simulation parameters for the architecture we explore in this paper. We include an 8K entry gshare branch predictor in our model. In addition to modeling all of the structures and latencies in the architecture, we have extended SimpleScalar to include a cycle accurate, execution driven model of chip multiprocessing (CMP) [6]. All the parameters used in our multicore experiments are the same as in Table 1 for each core, except that we increase the size of the L2 cache to an 4MB, 4-way set-associative cache shared among all cores.

Per-thread performance metrics are measured for execution up to a maximum per-thread instruction count. All completed threads continue execution past this point while other threads execute. This prevents freeing of resources when certain threads complete earlier than others.

We make use of *weighted speedup* [18] as a performance measure. This metric ensures that high IPC threads are not favored and that we are measuring real increases in the rate of progress of all applications in the mix. Weighted speedup equalizes the contribution of each thread to the sum of total work completed in the interval by dividing the IPC of that job in the mix by the IPC of a single threaded run.

# 4 Tuning a Single Core

While there are many different papers that explore the best way to take a single architectural feature and modify it to strike a balance between latency and capacity, there is a shortage of work that examines what the ramifications are when *most* of the structures are built in this way. As our goal is to develop techniques to make good choices between a variety of different run-time processor configurations, we need to begin with a processor model that has been highly decoupled. Figure 1 illustrates the architecture we explore in this study, with many previously proposed helper engines decoupled from the core pipeline. In this section we limit ourselves to single core designs, and then in Section 5 we show how to extend the ideas developed here to apply to cases where multiple cores may compete for resources.

## 4.1 Helper Engines

Because there is no existing simulation infrastructure with built-in decoupled helper engines, we modified our CMP-extended version of SimpleScalar [4] to include a variety of proposed designs. The helper engines we include, when combined together, decouple all of the major pieces of the modern processor, and we believe that they represent a realistic vision of the next generation of aggressive superscalar designs. This section includes a description of our target processor core and the helper engines that assist it.

### 4.1.1 Data Cache

We consider a small L0 cache in our core, as in [8]. Our L1 data cache helper engine extends the cache hierarchy, providing larger capacity than the L0 at a faster latency than our L2. While data caches are perhaps the most well studied part of the processor, we will later show the benefits of controlling the data cache configuration in conjunction with other helper engines.

### 4.1.2 Instruction Fetch

Similar to the data cache, we make use of a smaller L0 instruction cache and an L1 instruction cache helper engine. To compensate for the smaller cache size, we use out-of-order instruction fetch as described in [21]. In this scheme, a placeholder is used in the instruction fetch queue (IFQ) to maintain program order - and the execution core stalls if the next entry to be consumed from the IFQ is still in flight. We model the complexity this brings to the IFQ by implementing the equivalent of an MSHR [9] for the instruction cache.

### 4.1.3 Data Prefetch

We model a stream buffer architecture guided by a stride-filtered markov predictor as proposed in [14]. The stream buffers of the prefetch helper engine are *only* accessed on L0 data cache misses. We allow a single prediction and a single

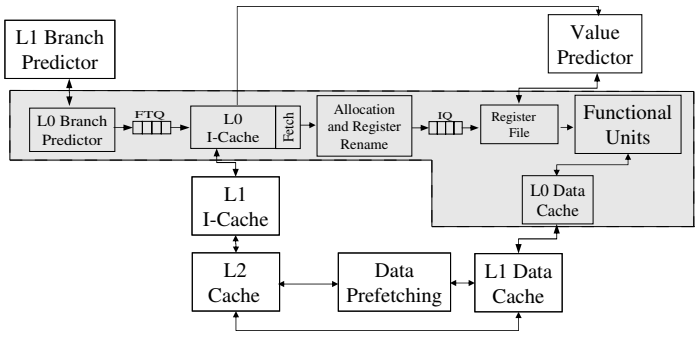


*Figure 1: The single core architecture explored in this study.*

prefetch per cycle, guided by the address predictor trained on the L0 miss stream.

#### 4.1.4 Value Prediction

Value predictionis one approach to break true data dependencies and create more instruction level parallelism in an application. We use a hybrid value predictor [23] to predict load instructions only. This structure can be accessed early in the pipeline as we only need the PC of the instruction to make the prediction. Our value predictor helper engine is limited to two predictions per cycle. We make use of an extra bit associated with each instruction in the instruction cache to dynamically mark instructions for value prediction. In this study, we only mark instructions that are loads.

#### 4.1.5 Branch Address Prediction

Our architecture makes use of a basic block target buffer (BBTB) [24], a branch address predictor that predicts an entire basic block each cycle. The PC at the head of the basic block serves as an index to the predictor, which returns a target address, a fallthrough address, a branch type, and two per-branch prediction counters: one to make per-branch direction predictions, and one to arbitrate between the per-branch prediction and the global branch direction prediction.

Our architecture makes use of a small first level BBTB and a second level BBTB helper engine, similar to [13]. Similarly we decouple branch prediction from the instruction cache using a fetch target queue (FTQ) [13]. On a first level BBTB miss, the BBTB helper engine is probed and fetch stalls until a response is received from the helper engine. If the helper engine also misses, we guess a fixed fetch block size and continue fetching until a misprediction is detected.

### 4.2 Helper Engine Utilization

While there are many circuit level advantages to decoupling large structures from the processor core, it also makes it very easy to *tune the processor to the specific needs of an application*. Each of these engines has a well defined interface, and the processor can tolerate a wide range of access latencies to any one of these engines (further explored in Section 5). Thus, adding supplemental control to each of these devices that will allow them to maintain separate power states does not introduce unreasonable cost or performance degradation.

As might be expected, for any given program some helpers will be more helpful than others. If the program is spending all of its time doing data accesses, it is more likely to get benefit from the data cache and prefetcher than the instruction cache. In a naive design, one might leave all of the helper engines in an "on" state at all times. Clearly, this will be wasteful if a program gets no benefit from a subset of the helper engines. In fact, we have found that up to half of the helper engines can be turned off at any given point in time, with almost no performance impact (2%). The problem is knowing *which* engines to turn off.

Note that there is an overhead associated with power gating, and the coarse-grain organization of these auxiliary structures into helper engines provides a useful abstraction for power gating these structures. Moreover, phase-based optimization helps to hide the latency associated with power gating, since the granularity of our helper allocations is much larger than the latency of power gating.

To further illustrate this, we plotted the set of critical helper engines that are needed to maintain top performance for each program in Figure 2. Figure 2 shows the minimal set of helpers needed to achieve maximal performance for the top four phases of 11 of the programs we have examined (shown in the rows). The columns of the table represent the helper engines, (d=data cache, p=prefetching, b=branch prediction, i=instruction cache, and v=value prediction). An X in a given square indicates for that phase, the corresponding helper engine should be turned "on". Helpers without X's can be disabled without affecting performance. We refer to setting the helper engines "on" or "off" as the configuration of the helpers in one of two power states.

All of the configurations in Figure 2 perform within 5% of the configuration shown to have the absolute highest performance (typically the configuration with all helper engines active). The last two columns present the speed up of this configuration relative to best and worst configurations (all helpers on, and all helpers off). These configurations were found strictly by a brute force search of the design space, simulating every possible configuration and taking the configuration with the least number of helper engines "on" that was still within 5% of the case with all helper engines turned on.

Obviously, trying each of the $2^n$ possible configurations is not a viable design choice for a runtime system, but there are several important points that can be drawn from this graph. It is clear from Figure 2 that there is no *one* good configuration that fits all applications. For many programs different configurations are even needed for different phases. This means a new, intelligent, and adaptive management scheme is going to be needed.

| | | d | p | b | I | v | S U /all | S U /none |
|---|---|---|---|---|---|---|---|---|
| applu | 1 | | X | | | | -4% | 9% |
| | 2 | | | | | | -2% | 0% |
| | 3 | | X | | | | -4% | 20% |
| | 4 | | X | | X | | 0% | 19% |
| apsi | 1 | | | X | X | | -3% | 40% |
| | 2 | | | | | | 0% | 2% |
| | 3 | | | | | | -4% | 0% |
| | 4 | | | | | | -4% | 0% |
| art | 1 | | X | | | X | 0% | 16% |
| | 2 | | X | | | | -4% | 8% |
| | 3 | | | | | X | -4% | 7% |
| | 4 | | X | | | | -4% | 8% |
| bzip2 | 1 | | X | | | X | 0% | 23% |
| | 2 | | | | | X | -2% | 8% |
| | 3 | | X | | | X | 0% | 23% |
| | 4 | | X | | | X | -3% | 10% |
| crafty | 1 | X | | X | X | | -4% | 174% |
| | 2 | X | | X | X | | -3% | 138% |
| | 3 | X | | X | X | | -2% | 192% |
| | 4 | X | | X | X | | -4% | 156% |
| eon | 1 | X | | X | X | | -2% | 125% |
| | 2 | X | | X | X | | -1% | 110% |
| | 3 | X | | X | X | | -2% | 107% |
| | 4 | X | | X | X | | -1% | 105% |

| | | d | p | b | I | v | S U /all | S U /none |
|---|---|---|---|---|---|---|---|---|
| galgel | 1 | | | | | | -4% | 0% |
| | 2 | | | | | | -1% | 0% |
| | 3 | | | | | | -1% | 0% |
| | 4 | | | | | | -3% | 0% |
| gap | 1 | | | X | X | X | -3% | 174% |
| | 2 | | | X | X | | -4% | 40% |
| | 3 | | | X | X | | -4% | 36% |
| | 4 | | X | | | X | 0% | 85% |
| mcf | 1 | | X | | | X | -1% | 985% |
| | 2 | | | | | X | -2% | 15% |
| | 3 | | | | | X | -3% | 407% |
| | 4 | | | | | X | -2% | 10% |
| mesa | 1 | | | X | X | X | -1% | 109% |
| | 2 | | | X | X | X | -1% | 94% |
| | 3 | | | X | X | X | -1% | 101% |
| | 4 | | | X | X | X | -1% | 102% |
| parser | 1 | X | | X | | X | -2% | 29% |
| | 2 | X | | X | | X | -1% | 24% |
| | 3 | X | | X | | | -4% | 17% |
| | 4 | X | | X | | X | -1% | 18% |

*Figure 2: Optimal helper configurations for top executed phases of benchmarks*

### 4.3 Helper Configuration

Helper engine configuration could be tackled in a variety of different ways. While static approaches could use profile or compiler guided heuristics to find one good configuration for the application as a whole, this may prove ineffective due to the time varying behavior of applications. A more effective static approach may be to inject special reconfiguration instructions that help tune the processor to specific phases. While these techniques are possible, in this paper we choose to focus on hardware based dynamic techniques as they require no a priori knowledge of the program and operate in a completely on-line manner.

To guide our decision about which configuration to choose, we need some information on how a program interacts with the processor. This information must be simple and cheap to obtain, and should be highly indicative of the benefits the program is reaping from access to each helper engine. In order to gather this information, we use simple performance counters that track the "help" that each engine provides. For example, performance counters can track the number of hits in cache helper engines, successful predictions by predictor helper engines, and so forth. Specifically, our helper engines are modified to track the following events:

**Data/Instruction Cache:** the number of times a cache line that missed in the L0 data/instruction cache hits in the data/instruction cache helper engine.

**BBTB:** the number of times a PC that missed in the L0 BBTB hits in the BBTB helper engine.

**Data Prefetcher:** the total number of hits in the stream buffers and in the L2 cache that were brought in by the prefetcher. Each line in the L2 cache is augmented with a bit to indicate whether it was brought in by a demand miss or a prefetch. On the first use of a line marked as a prefetch, the bit is flipped.

**Value Prediction:** the number of instructions issued using a predicted value.

These counter values are compared to pre-determined threshold values to decide whether the helper engine should be turned on or off. We performed a sensitivity analysis to various threshold values for the SPEC 2000 benchmark suite with reference inputs. We show results for a conservative set of thresholds to curtail performance degradation. More aggressive thresholds can be used when power reduction is the primary objective. Furthermore, dynamic threshold values can be calculated by adjusting the thresholds to maintain certain "on" and "off" state performance counter standard deviation values. However, this is beyond the scope of this paper, and we leave this for future work.

We make use of phase-based memoization [15] to track the helper engine configuration per application phase. A small hardware structure tracks a bit vector per phase for all the helpers in the architecture. If the bit at a particular location is set, the helper represented by that particular location should be on. Otherwise the helper can be turned off. The first time a phase is seen, we turn all helpers on and track the performance using the above counters. Each counter is compared against the threshold for keeping the helper engine on, and the bit vector for that particular phase is updated in our phase-based memoization table. The helper engines can then be guided by a simple last phase predictor. The sampling period need not be as long as a phase, and can be limited to an interval of one million cycles.


*Figure 3: Comparison of the configuration from Figure 2 (light grey) and our counter guided configuration (dark grey).*

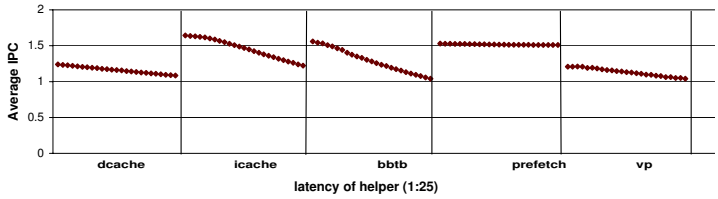


*Figure 4: Impact of Latency on Our Helper Engines*

We also memoize the observed IPC for each phase during the sampling period, and if the IPC resulting from a particular configuration is not within some threshold IPC seen during sampling (5% for this study), we clear the bit vector for that phase and force another sampling of the above counters. This helps recover from phase mispredictions or events that can impact performance (such as power throttling).

### 4.4 Performance of Counter-Guided Configuration

Figure 3 compares the performance of the configuration from Figure 2 (light grey) to our counter-guided configuration. Performance here is normalized to an architecture running with all helper engines on. On average, we are able to come within 1.5% of that performance with an average of only 2.6 helpers turned on. The configuration from Figure 2 comes within 2% of the performance of all helpers on, only using 2.2 helpers on average. In some cases, we use one more helper engine than needed due to our choice of conservative thresholds. More often than not this is the data cache helper engine, which can provide load hits that are not performance critical. One approach to fine tuning this further would be to incorporate the notion of load criticality [20] or to try and correlate this with the number of L2 misses – applications with a large number of L2 misses may not see benefit from an increase in L1 hits if the L2 misses that dominate the critical path of the application are not reduced.

These results demonstrate the ability of performance counters to fine tune helper engine utilization. This can allow an architecture to reduce power wasted in helper engines that do not provide useful work or as we will see in the next section, to coordinate sharing among cores using a common pool of helper engines.

## 5 Management Across Cores

As demonstrated in Section 4, resource demand varies significantly across different applications and even across different phases of the same application. In the case of a single core, this fact can be exploited to reduce power by finding a configuration that still provides good performance but with a bare minimum of helper engines left in a high power state. Managing helper engines when multiple cores are involved presents a tougher challenge. While the easiest approach to supporting multiple cores on a chip would be to give each one its own set of helper engines, previous work has shown that this instills unnecessary area complexity without a significant performance benefit [5, 11]. The alternative that we also explore in this paper is the use of a common pool of helpers, shared among all the cores. Sharing has a number of benefits such as reducing the area spent to implement redundant functionality and the potential of optimization through dynamic resource allocation. In this section, we present techniques to effectively manage helper engines in a multicore environment.

### 5.1 Design Decisions

There are a large number of design decisions to be made in examining helper engines in a multicore setting: the number of cores, the number of each type of helper engine, the topology of the interconnect between cores and helper engines, the physical layout of cores and helpers, and the application mix to execute on the cores. This is an enormous design space, and it is simply not manageable to try all possible combinations. To get a set of experiments which is tractable, we limit our search in this paper by considering results for two possible machine organizations: four cores sharing a single helper engine of each type and four cores sharing two helper engines of each type. Additionally, we consider applications that are not cooperative, but our work could certainly be applied to cooperative multithreading. We assume that all cores share a common second level cache, and that any core may connect to any helper engine.

One question that immediately comes to mind when we propose that any core may connect to any helper engine is that it is going to take a good deal longer to communicate with a helper engine on the other side of the chip than with one in close physical proximity to the core. While this is true when partitioning arbitrary processor resources, helper engines have an inherent advantage due to their higher latency tolerance.

To demonstrate that helper engines are naturally latency tolerant, we present Figure 4 which plots the performance impact of latency on our various helper engines by varying the access latency from 1-25 cycles. For these results, we consider a single core with private helpers, and average the IPC observed over benchmarks that use the helper from table 2. As seen from the figure, for most helpers, there is little impact on performance of these helpers from smaller latencies – the most is seen by the branch target buffer (BBTB) helper, which suffers an IPC degradation around 1% for each additional cycle of latency. Prefetching sees the least impact, less than a 0.01% drop in IPC for each additional cycle of latency. The prefetcher hides the latency of memory, and even 25 cycles is tolerable

when compared with this latency. The remaining helpers see less than 0.5% degradation per cycle for each additional cycles. As the latency increases above 10 cycles, its impact on the performance of the helper increases non-linearly.

Our architecture is also not impacted by nonuniform access latency from different cores to a common helper. The arbiter that selects what requests from a core should be serviced by a helper would be located close to the helper itself. Therefore, if core A sees a two cycle latency to a helper and core B sees a single cycle latency to a helper, and if core A pipelines its requests over two cycles, then the helper will simply see a stream of requests from A and B without any notion of heterogeneous latency. A will see its predictions a cycle later than B will see its predictions, but as we have demonstrated, this has a negigible impact on performance.

Our multicore architecture and its floorplan are shown in Figure 5. We use CACTI to estimate the size and dimensions of the L2 cache and all of our helpers. The area of our core was calculated using the area of EV6 and EV5 scaled to a *70nm* feature size, similar to [10].

Our flexible sharing requires a link from each core to each helper. This would double the number of interconnections compared to a conjoined architecture where each helper is statically shared between two cores. We used a similar method of crossbar area estimation as [11]. For our choice of helpers and their respective bandwidth requirements, the crossbar area occupies 10% of the total area of the processor, which is 5% more than conjoined cores.

The other hidden cost in sharing helpers is the potential increase in requests for each resource, making helper engine bandwidth a serious concern. Ideally, each core would have its own dedicated port to each helper engine, but the cost would be prohibitive. Instead, a helper can make use of port arbitration to satisfy multiple core requests. One possibility is allowing cores to take turns accessing a helper. Another would involve more sophisticated control hardware that would arbitrate among several requests, much like what is done with a unified second level cache.

In an architecture where cores share a common pool of helpers, the helpers can either be exclusively assigned to a core (i.e. partition the helpers), shared among several cores (i.e. sharing common helpers), or some combination of both (i.e. some of the cores sharing a common helper). Partitioning is useful when the bandwidth or internal storage demands placed on a helper by a single core would preclude benign sharing with other cores. In that case, the most favorable option would be dedicating a helper to a single core. Sharing can be useful when individual cores do not consume all of the bandwidth or storage space of a given helper.

### 5.1.1 Always On vs. On Demand Sharing

Different helper engines exhibit different tolerances to sharing. At a high level, helpers can be divided into those that are accessed on-demand and those that are always active. On-demand helpers include those that are hierarchical extensions

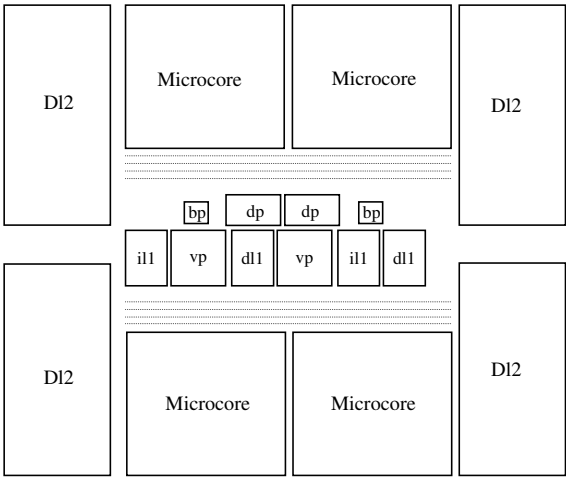


*Figure 5: Our Multicore die floorplan, with 2 set of helpers in the middle of the cluster, and microcores and L2 cache banks distributed around the outside.*

of core structures, like the instruction cache, data cache, and branch predictor. These helpers are only accessed when their corresponding core structure misses. Locality in the corresponding core structures filters the majority of the requests obviating frequent accesses to these helpers. This means that bandwidth to the helper may more easily be shared among multiple cores. However, the amount of state contained in the helper may still be insufficient to allow effective sharing between cores – but bandwidth is not usually a limiting factor.

Other helpers do not have corresponding core structures, and are therefore not on-demand. The value prediction and prefetching helpers are examples of this class of helper engines. Any load instruction can be value predicted and any cache miss can initiate a prefetch stream, but a helper may not have enough bandwidth to handle competing requests from multiple cores if there are lots of loads or cache misses. Helper state is also a problem here, as sharing cores would need to contend with one another for value and address predictor space.

### 5.1.2 Simple vs. Counter Guided Sharing

One simple approach to sharing is to have conjoined cores take turns accessing a common set of helper engines. Taking turns is a viable option for on-demand helpers where contention for helper bandwidth is less common. But helpers that are not on-demand can suffer from a turn-based approach. Consider value prediction in a two core setting, where each cycle a new set of load PCs are fetched and could potentially be value predicted. Taking turns would mean that value prediction would only occur every other cycle, potentially halving the number of value predicted instructions. Note that the issue here is not access latency (because helper engines are latency tolerant), but the fact that we lose the opportunity to predict load instructions. This problem only becomes worse when trying to share among more than two cores. We further optimize the

turn-based strategy by allowing other cores to access a helper engine on a given core's turn if that core does not require the helper's bandwidth. This can happen when a core has suffered a pipeline flush or in the case of on-demand helpers that simply do not see any accesses to the helper.

Even though some cores may make use of a helper, the helper may not provide any benefit, even in the case of on-demand helpers. Consider a thread whose working set does not fit in the data cache helper engine – it will thrash in the data cache as it tries to contain its working set, but will still be plagued by misses that must be serviced by the L2 and will evict potentially useful entries from other threads. Similarly, a thread may not see any benefit from value prediction and may simply be stealing available bandwidth from a thread that does see benefit.

We make use of the utilization counters from Section 4 to guide helper engine sharing. These counters provide a filtering mechanism to avoid sharing a helper among cores that see no benefit from that helper. Cores can then request access from a global helper arbiter to the helpers from which they expect to see benefit.

While filtering useless sharing is vital, we also need to provide an intelligent approach to choose what cores will share a common helper engine. In a four core scenario where all cores (labeled A-D) want a BBTB helper engine, and there are only two helpers available, performance may substantially improve if A and B are allowed to share instead of A and C. Or, it may be best for A to have its own private helper and for B-D to share the remaining helper.

To arbitrate sharing among the filtered set of helper engines, we make use of correlating counters that can guide sharing. A good example of this is the prefetching helper engine. If multiple cores are contending for a pool of prefetch helper engines, threads that have a greater magnitude of prefetches should be given private access to a prefetcher if one is available. If there are not enough prefetchers to grant a private engine to threads with a large number of prefetches, threads with a comparable number of prefetches should be paired together. This prevents threads with a greater number of prefetches from starving other threads from getting access to stream buffers.

At each core, the correlating counters are tested against a number of thresholds to classify the demand for a particular helper into a utilization class (i.e. light, medium, heavy). The utilization class is then memoized in the phase-detection hardware to track the expected utilization class of each helper at each phase.

The global helper arbiter is responsible for taking the requests from all cores for helpers (this has already been filtered at each core by the performance counters and phase-detection mechanisms) and the actual utilization class for each helper requested from each core. The global arbiter then tries to match requests of similar utilization classes together in the case of helpers that are not on-demand and tries to mix requests in the case of helpers that are on-demand.

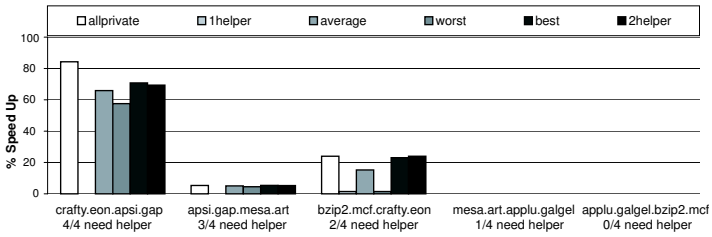


*Figure 6: Simple vs Counter-Guided Sharing of the BBTB Helper Engine*

## 5.2 Sharing Results

To better evaluate the difference between the two types of helper engines and the two types of sharing, we more closely examine the BBTB helper engine (an on-demand helper) and the prefetcher helper engine (not on-demand). For each, we examine situations where four cores have private versions of all other helper engines but either the BBTB or the prefetcher. The four cores are then forced to share either one or two instances of either the BBTB or the prefetcher. For the results in this section, an architecture where four cores are sharing a single instance of a helper is denoted with a *-1h* and an architecture where four cores are sharing two helper engines is denoted with a *-2h*. We consider conjoined sharing (*conjoined*) and counter guided sharing (*counter*), as well as the case where all cores have their own private engines (*private*). Four conjoined cores can share one helper in only one way (all four sharing the one helper), but four cores can be conjoined to share two helpers in three ways: core A and B together, core A and C together, or core A and D together. To capture the variation in performance possible depending on how cores are conjoined, we consider three bars for conjoined cores: *best-conjoined* (the best case combination of cores), *worst-conjoined* (the worst case combination of cores), and *avg-conjoined* (the average performance across all combinations). Note that there is no one combination that is always best or worst - the best and worst cases vary dramatically from helper to helper and from application to application.

For each helper that we explore, we construct an application mix by selecting five benchmarks that benefited from the helper engine and five benchmarks that did not benefit from the helper engine. We then form five application mixes of four threads each that represent all possible combinations: all threads need the helper, three out of four threads need the helper, two out of four threads need the helper, only one thread needs the helper and no thread needs the helper – in that order. For cases where only one or no application demands a helper engine, there is obviously little impact from these sharing approaches – but these results are shown for completeness.

Our helper configurations are as presented in Section 3. The value predictor helper engine has only two ports, and can therefore only satisfy two requests for prediction per cycle. The prefetcher can only prefetch one cache line per cycle.

Figure 6 illustrates the performance of an on-demand helper: the BBTB helper engine. For the single helper runs, there is not much improvement from counter-guided sharing

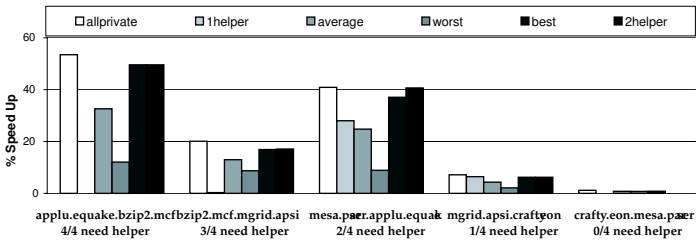


*Figure 7: Simple vs Counter-Guided Sharing of the Prefetching Helper Engine*

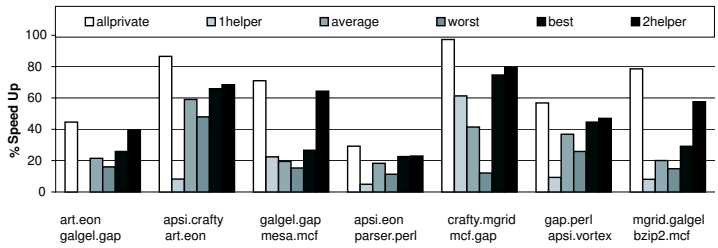


*Figure 8: Simple vs Counter-Guided Sharing of All Helper Engines*

because on-demand sharing already inherently filters requests (counter based-1 helper). Intelligent sharing in this case would be beneficial if it can eliminate useless accesses to the helper engine. On-demand helpers naturally will not be accessed if there is no benefit. However, when sharing two helpers among four cores, there is a destructive relationship between `eon` and `crafty` when sharing a common BBTB due to aliasing. Our counter-guided approach is able to determine the best combination for benign sharing – even when all four applications want to share the BBTB. Because `eon` and `crafty` see many more helper engine BBTB hits than other applications, they should not be combined together to avoid contention for space in the BBTB. The gap in performance between having a single BBTB helper and having two BBTB helpers is clearly greater when more applications demand the BBTB, demonstrating the impact that contention for space can have, even for on-demand helpers. The last two mixes of benchmarks, where only one or no application needs the BBTB, do not see any impact from sharing approaches.

Figure 7 illustrates the performance of a helper that is not on-demand: the prefetch helper engine. Counter-guided sharing is useful for the single helper run when there are two or three benchmarks competing for bandwidth that do not see any benefit from the helper. The benchmark mix `mesa-parser-applu-equake` is such a case, where `mesa` and `parser` do not benefit from prefetching but `applu` and `equake` do. By filtering `mesa` and `parser` out via our correlation counters, we are able to use a single helper engine to outperform the worst and average cases of conjoined cores with two helpers.

With two helpers, there is more disparity between different conjoined core runs. `Applu` and `mcf` issue far more prefetches than `bzip2` and `equake`, and unlike the case of on-demand helpers, it is actually beneficial here to have `applu` and `mcf` share the same helper engine. If we combine either one of these applications with one that benefits from prefetching but does not have the same magnitude of prefetches (like `bzip2` or `equake`), the application with less prefetches will not get a fair share of stream buffer resources. In the case of `mesa-parser-applu-equake`, `parser` contends for prefetch bandwidth, despite not seeing a benefit from prefetching, and impedes the prefetching of `applu` or `equake`. The counter-guided approach is able to identify the inability of `parser` to effectively use prefetching, and prevents this degradation. For all runs, our counter-guided approach is able to perform as well or better than the best conjoined core combination.

Figure 8 presents results for sharing all helpers at once. The first application mix on the figure enjoys a large improvement from our counter-guided approach for two main reasons. First, `gap` is able to get a private value predictor. Second, `art` and `eon` do not do well when conjoined with `galgel`. By giving flexibility to helpers in how they share, our approach is able to outperform any conjoined combination. This is also evident from the third benchmark mix, where value prediction contention hampers the performance of `mcf` and `gap`. On average our counter-guided approach for one set of helpers sees 13% improvement over a baseline naively sharing helpers among all cores. Sharing two set of helpers, our approach provides 54% improvement while conjoining cores can see benefit ranging from 20% to 40% depending on how cores are conjoined.

### 5.2.1 Constructive Sharing

As we demonstrated in the previous section, sharing one helper among four cores, can degrade performance significantly when applications running on all cores need the helper. However, there are cases where common code or data (i.e. OLTP, parallel processing) may be executing in the CMP environment, and sharing can actually be constructive if threads are allowed to share state in a common helper engine. Our flexible helper management allows constructive sharing of a helper among such workloads. We consider the case where the same application is executing on all cores, but each application instance is executing different input sets or different phases of the same input set.

Figure 9 illustrates the benefit of constructive sharing for the BBTB and instruction cache helpers. For `crafty`, `eon` and `mesa` we simulated four different phases of each application running concurrently on four cores. For `vortex` we used four different inputs. All of these application use both BBTB and instruction cache helpers intensively. The first bar shows results when there are private helpers dedicated to each core, and the second bar shows results when only one helper is shared constructively among the four cores. The speedup presented is relative to one helper shared among four cores, but without any constructive sharing among threads. Our counter mechanism is still used in this case. Our results indicate that sharing the instruction cache between multiple cores incurs a similar miss rate as a dedicated cache per core with the same capacity, as reported in [12]. In `vortex`, a constructively

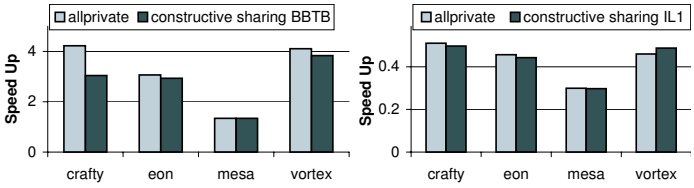


*Figure 9: Speedup of private helpers, and constructively shared helpers relative to the performance of sharing helpers destructively*

shared instruction cache even outperforms the private cache performance by avoiding misses to cache blocks used by multiple cores. The contention when sharing the BBTB among applications that have high demand for this helper can be extremely severe when not shared constructively. `Crafty` and `vortex` see a 4X speedup when using dedicated BBTB helper engines instead of sharing a single BBTB helper. This is due to the reduced accuracy of branch prediction when threads thrash for space in the shared predictor. There is a significant increase in the number of instructions executed as a result of this mispeculation which can further pollute caches and waste energy. Sharing the BBTB constructively among cores eliminates this thrashing effect for all applications simulated except `crafty` which still sees some impact from this. This application has more complex branch behavior that can inhibit constructive sharing across different phases.

## 6 Summary

In this paper we explore helper engine management policies, both for a single core and in a multicore environment. Cores with decoupled helper engines provide an opportunity to dynamically tune processor resources on an application phase basis. With our counter-guided helper management, we can come within 2% of the best performing helper engine configuration (typically the one with all five helper engines active), but with an average of less than three helpers turned on. This counter-guided scheme can be applied to the multicore environment to more effectively share a pool of common helper engines among a number of cores. Our approach to sharing is intelligent and flexible enough when used with four cores sharing two sets of helpers to see an average 54% weighted speedup over a baseline naively sharing only one set of helpers. Statically shared helpers can see benefit ranging from an average of 20% to 40% depending on how cores are shared. Constructive sharing can provide even more benefit, effectively providing performance comparable to private helper engines when running the same application on all cores, even for different inputs and phases.

## References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[2] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[3] D. Brooks, P. Cook, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, and M. Gupta. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. In *IEEE Micro*, November 2000.

[4] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[5] R. Dolbeau and A. Seznec. Cash: Revisiting hardware sharing in single-chip parallel processor. Technical Report IRISA Report 1491, IRISA, November 2002.

[6] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30, 1997.

[7] H-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 71–81, June 2002.

[8] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache:an energy efficient memory structure. In *IEEE International Symposium on Microarchitecture*, December 1997.

[9] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium of Computer Architecture*, pages 81–87, May 1981.

[10] R. Kumar, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *36th International Symposium on Microarchitecture*, December 2003.

[11] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-core chip multiprocessing. In *37th International Symposium on Microarchitecture*, December 2004.

[12] Partha Kundu, Murali Annavaram, Trung Diep, and John Shen. A case for shared instruction cache on chip multiprocessors running oltp. *SIGARCH Comput. Archit. News*, 32(3):11–18, 2004.

[13] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[14] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[16] P. Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *Technical Report*, 2001.

[17] J. E. Smith. Instruction-level distributed processing. *IEEE Computer*, 34(4):59–65, April 2001.

[18] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[19] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *29th Annual International Symposium on Computer Architecture*, 2002.

[20] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[21] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *30th International Symposium on Microarchitecture*, pages 34–43, December 1997.

[22] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[23] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, pages 281–290, December 1997.

[24] T. Yeh and Y. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, December 1992.