In Proceedings of the 31st International Symposium on Computer Architecture, June, 2004

Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance

Rakesh Kumar[†], Dean M. Tullsen[†], Parthasarathy Ranganathan[‡], Norman P. Jouppi[‡], Keith I. Farkas[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

[‡]HP Labs
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

A single-ISA heterogeneous multi-core architecture is a chip multiprocessor composed of cores of varying size, performance, and complexity. This paper demonstrates that this architecture can provide significantly higher performance in the same area than a conventional chip multiprocessor. It does so by matching the various jobs of a diverse workload to the various cores. This type of architecture covers a spectrum of workloads particularly well, providing high single-thread performance when thread parallelism is low, and high throughput when thread parallelism is high.

This paper examines two such architectures in detail, demonstrating dynamic core assignment policies that provide significant performance gains over naive assignment, and even outperform the best static assignment. It examines policies for heterogeneous architectures both with and without multithreading cores. One heterogeneous architecture we examine outperforms the comparable-area homogeneous architecture by up to 63%, and our best core assignment strategy achieves up to 31% speedup over a naive policy.

1 Introduction

The design of a microprocessor that meets the needs of today's multi-programmed compute environment must balance the competing objectives of high throughput and good single-thread performance. To date, these objectives have been addressed by adding features to monolithic superscalar processors to increase throughput at the cost of increased complexity and design time. One such feature is simultaneous multithreading [24, 23] (SMT). An alternative approach has been to build chip multiprocessors [8, 11] (CMPs) comprising multiple copies of increasingly complex cores.

In this paper, we explore an alternate design point between these two approaches, namely, CMPs comprising a heterogeneous set of processor cores all of which can execute the same ISA. The heterogeneity of the cores comes from differences in their raw execution bandwidth (superscalar width), cache sizes, and other fundamental characteristics (e.g., in-order vs. out-of-order). This architecture has been proposed and evaluated in earlier work [13, 14] as a means to increasing the energy efficiency of single applications. However, as we demonstrate in this paper, the same architecture may be used to deliver greater throughput and improved area efficiency (throughput per unit area) without significantly impacting single-thread performance.

We evaluate a variety of heterogeneous architectural designs, including processor cores that are themselves multithreaded, an extension to the original architecture proposal [14]. Through this evaluation, we make the following two contributions.

First, we demonstrate that this approach can provide significant performance advantages for a multiprogrammed workload over homogeneous chip-multiprocessors. We show that this advantage is realized for two reasons. First, a heterogeneous multi-core architecture has the ability to match each application to the core best suited to meet its performance demands. Second, it can provide better area-efficient coverage of the whole spectrum of workload demands that may be seen in a real machine, from low thread-level parallelism (providing low latency for few applications on powerful cores) to high thread-level parallelism (where a large number of applications can be hosted at once on simple cores).

Overall, our representative heterogeneous processor using two core types achieves as much as 63% performance improvement over an equivalent-area homogeneous processor. Over a range of moderate load levels (e.g., 5-8 threads), we see an average gain of 29%. For an open system with random job arrivals, the heterogeneous architecture has much lower average response time over a range of job arrival rates and remains stable for arrival rates 43% higher than that for which a homogeneous architecture breaks down.

Our second contribution is to demonstrate dynamic thread-to-core assignment policies that realize most of the potential performance gain. These policies significantly outperform a random schedule, and even beat the best static assignment (using hindsight) of jobs to cores. These heuristics match the diversity of the workload resource requirements to the cores by changing the workload-to-core mapping either periodically or in response to triggering events. We study the design space of job assignment policies, examining sampling frequency and duration, and how core assignment is made. Our best policy outperforms naive core assignment by 31%.

We also study the application of these mechanisms to the cores in a heterogeneous processor that includes multithreaded cores. Despite the additional scheduling complexity posed by the simultaneous multithreading cores (due to an explosion in the possible assignment permutations), we demonstrate the existence of effective assignment policies. With these policies, this architecture provides even better coverage of a spectrum of load levels. It provides both the low latency of powerful processors at low threading levels, but is also comparable to a large array of small processors at high thread occupancy.

The rest of the paper is organized as follows. Section 2 motivates heterogeneous design for performance. Section 3 describes our measurement methodology. Section 4 discusses the performance benefits from our architecture and our scheduling approaches to solve the new design issues associated with these architectures. Section 5 concludes.

2 Architecture and Background

This section illustrates the potential benefits from architectural heterogeneity, introduces issues in using core heterogeneity with multi-programmed workloads, and discusses prior related work.

2.1 Exploring the potential from heterogeneity

The advantages of heterogeneous architectures stem from two sources. The first advantage results from more *efficient adaptation to application diversity*. Applications (or different phases of a single application) place different demands on different architectures, stemming from the nature of the computation [14]. While some applications take good advantage of the most advanced processors, others often under-utilize that hardware and suffer little performance loss when run on a less aggressive processor. For example, a floating-point application with regular code might make good use of an out-of-order pipeline with high issue-width; however, a bandwidth-bound, control-sensitive application might perform almost as well on an in-order core with low issue-width. Given a set of diverse applications and heterogeneous cores, we can assign applications (or phases of applications) to cores such that those that benefit the most from complex cores are assigned to them, while those that benefit little from complex cores are assigned to smaller, simpler cores. This allows us to approach the performance of an architecture with a larger number of complex cores.

The second advantage from heterogeneity results from a more *efficient use of die area for a given thread-level parallelism*. Successive generations of microprocessors have been obtaining diminishing performance returns per chip area. This is evident from the following. Microprocessor implementation technology has been scaling for many years according to Moore's Law [15] for lithography and roughly according to MOS scaling theory [5]. For a given $O(n)$ scaling of lithography, one can expect an equivalent $O(n)$ increase in transistor speed and an $O(n^2)$ increase in the number of transistors per unit area. If the increases in transistor speed and transistor count were to directly translate to performance, one would expect an $O(n^3)$ increase in performance. However, past microprocessor performance has only been increasing at an $O(n^2)$ rate [10, 9]. This is not too surprising, since the performance improvement of many microprocessor structures (e.g., cache memories) is less than linear with their size.

In an environment with large amounts of process or thread-level parallelism, such a nonlinear relationship between transistor count and microprocessor speed means that higher throughputs could be obtained by building a large number of small processors, rather than a small number of large processors. However, in practice the amount of process or thread level parallelism in most systems will vary with time. This implies that building chip-level multiprocessors with a mix of cores – some large cores with high single-thread performance and some small cores with high throughput per die area – is a potentially attractive approach.

To explore the potential from heterogeneity, we model a number of chip multiprocessing configurations that can be derived from combinations of two existing off-the-shelf processors from the Alpha architecture family – the EV5 (21164) and the EV6 (21264) processors. Figure 1 compares the various combinations in terms of their performance and their chip areas. In this figure, performance is that obtained from the best static mapping of applications to the processor cores. The staircase represents the maximum throughput obtainable using a homogeneous configuration for a given area.

We see from this graph that over a large portion of the graph the highest performance architecture for a given area limit, often by a significant margin, is a heterogeneous configuration. The increased throughput is due to increased number of contexts as well as improved processor utilization.

Constant-area comparisons do not tell the whole story, because equivalent area does not necessarily imply equal cost, power, or complexity. But the results are instructive, nonetheless. Note also that we explore a smaller subset of the design space than possible because of our constraint of focusing on just two generations of a commodity processor family. However, even with this limited approach, our results make a case for the general advantages for heterogeneity. For example, on the far left part of the graph, the area is insufficient to support a heterogeneous configuration of the EV6 and EV5 cores; however, our other data (not plotted here) on heterogeneous architectures using EV5 and EV4 (21064) cores confirm that these designs are superior in this region.

While the design of this architecture is likely to be very similar to a homogeneous CMP, designing the common interface to the shared cache, clock distribution, coherence, etc., may be somewhat more complex with multiple core types. It is also possible that verification costs might be higher if the diverse cores turn out to interact in less deterministic ways than in the homogeneous case. In this work, we recognize these is-

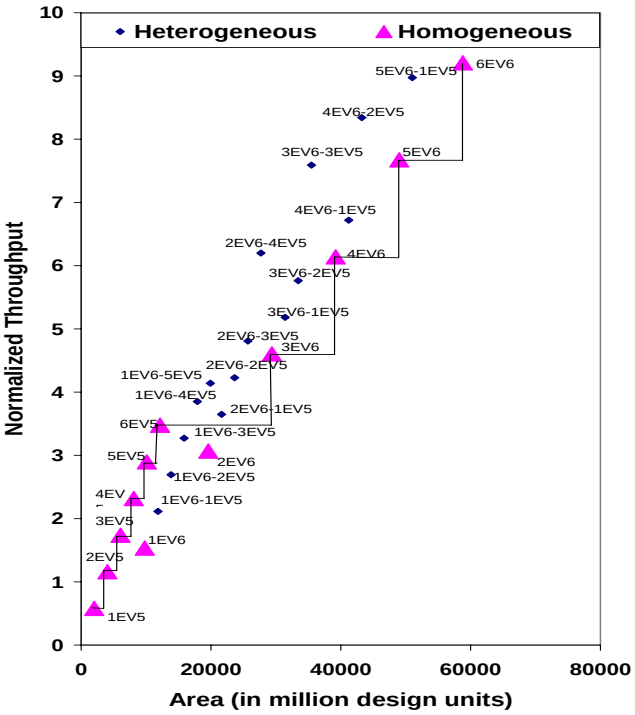

Figure 1. *Exploring the potential from heterogeneity. The points represent all possible CMP configurations using up to six cores from two successive cores in the Alpha family. The triangles represent homogeneous configurations and the diamonds represent the optimal static assignment on heterogeneous configurations.*

sues by limiting the core diversity to a small set of commodity processor cores from a single family.

The rest of this paper will examine particular points in this design space more carefully, allowing us to examine in detail the particular architectures, and how to achieve the best performance.

2.2 Prior work

Our earlier work [14] used a similar single-ISA heterogeneous multiprocessor architecture to reduce processor power. That approach moves a single-threaded workload to the most energy efficient core and shuts down the other cores to provide significant gains in power and energy efficiency. However, that approach provides neither increased overall performance nor increased area efficiency, the two goals of this research.

Heterogeneity has been previously used only at the scale of distributed shared memory multiprocessors (DSMs) or large computing systems [7, 16, 20, 3]. For example, Figueiredo and Fortes [7] focus on the use of a heterogeneous DSM architecture for the execution of a single parallel application at a time. They balance the workload by statically assigning multiple threads to the more powerful nodes, and using fast user-level thread switching to hide the context switch overheads of interleaving the jobs. Oh and Ha [16] presented a provably optimal (under certain simplifying conditions) static scheduling strategy for a heterogeneous DSM where each node can have a different execution time. They also incorporate the effects of interprocessor communication overhead. Banino *et al* [3] show how to determine the optimal steady-state scheduling strategy for allocating a large number of equal-sized tasks to a heterogeneous "grid" computing platform.

Given their significantly higher communication costs, previous work on DSMs has only statically exploited inter-thread diversity ([7] being the closest to our work). In contrast, heterogeneous multi-core architectures allow us to dynamically change the thread-to-core mapping to take advantage of changing phase characteristics of the threads/applications. Further, heterogeneous multi-core architectures can exploit fine-grained inter-thread diversity.

Other architectures also seek to address both single-thread latency and multiple-thread throughput. Simultaneous multithreading [24] can devote all processor resources on a superscalar processor to a single thread, or divide them among several. Processors such as the proposed Tarantula processor [6] include heterogeneity, but the individual cores are specialized toward specific workloads. Our approach differs from these in its use of heterogeneous commodity cores to exploit variations in thread-level parallelism and intra- and inter-application diversity for increased performance for a given chip area.

Prior work has also addressed the problem of phase detection in applications [17, 25] and task scheduling on multithreaded architectures [18]. Our work leverages that research, and complements these by demonstrating that phase detection can be used with task scheduling to match application diversity with core diversity for increased performance.

2.3 Supporting multi-programming

The primary issue when using heterogeneous cores for greater throughput is with the scheduling, or assignment, of jobs to particular cores. We assume a scheduler at the operating system level that has the ability to observe coarse-grain program behavior over particular intervals, and move jobs between cores. Since the phase lengths of applications are typically large [17], this enables the cost of core-switching to be piggybacked with the operating system context-switch overhead. Core-switching overheads are modeled in detail for the evaluations presented in this paper.

Workload-to-core mapping is a one-dimensional problem in [14] as the workload consists of a single running thread. With multiple jobs and multiple cores, the task here is not to find the best core for each application, but rather to find the best global assignment. All of our policies in this paper strive to maximize average performance gain over all applications in the workload. Fairness is not taken into consideration explicitly. All threads make good progress, but if further guarantees are needed, we assume the priorities of those threads that need performance guarantees will reflect that. The heterogeneous

Processor	EV5	EV6	EV6+
Issue-width	4	6 (OOO)	6 (OOO)
I-Cache	8KB, DM	64KB, 2-way	64KB, 2-way
D-Cache	8KB, DM	64KB, 2-way	64KB, 2-way
Branch Pred.	2K-gshare	hybrid 2-level	hybrid 2-level
Number of MSHRs	4	8	16
Number of threads	1	1	4
Area (in mm^2)	5.06	24.5	29.9

Table 1. *Configuration and area of the cores.*

architecture is also ideally suited to manage varied priority levels, but that advantage is not explored here.

An additional issue with heterogeneous multi-core architectures supporting multiple concurrently executing programs is cache coherence. In this paper, we study multi-programmed workloads with disjoint address spaces, so the particular cache coherence protocol is not an issue (even though we do model the writeback of dirty cache data during core-switching). However, when there are differences in cache line sizes and/or per-core protocols, the cache coherence protocol might need some redesign. We believe that even in those cases, cache coherence can be accomplished with minimal additional overhead.

3 Methodology

This section discusses the methodology we use for modelling and evaluating the various homogeneous and heterogeneous architectures. It also discusses the various classes of experiments that were done for our evaluations.

3.1 Hardware assumptions

Table 1 summarizes the configurations used for the cores in the study. As discussed earlier, we mainly focus on the EV5 (Alpha 21164) and the EV6 (Alpha 21264). For our experiments with heterogeneous multi-core architectures with multithreaded cores (discussed in Section 4.4), we also study a hypothetical multi-threaded version of the EV6 processor that we refer to as EV6+. All cores are assumed to be implemented in 0.10 micron technology and are clocked at 2.1 GHz (the EV6 frequency when scaled to 0.10 micron).

In addition to the individual L1 caches, all the cores share an on-chip 4MB, 4-way set-associative, 16-way L2 cache. The cache line size is 128 bytes. Each bank of the L2 cache has a memory controller and an associated RDRAM channel. The memory bus is assumed to be clocked at 533Mhz, with data being transferred on both edges of the clock for an effective frequency of 1GHz and an effective bandwith of 2GB/s per bank. Note that for any reasonable assumption about power and ground pins, the total number of pins that this memory organization would require would be well within the ITRS limits[1] for the cost/performance market. A fully-connected matrix crossbar interconnect is assumed between the cores and the L2 banks. All L2 banks can be accessed simultaneously, and bank conflicts are modelled. The access time is assumed to be 10 cycles. Memory latency was set to be 150 ns. We assume a snoopy bus-based MESI coherence protocol and model the writeback of dirty cache lines for every core-switch.

Table 1 also presents the area occupied by each core. These were computed using a methodology similar to that used in our earlier work [14]. As can be seen from the table, a single EV6 core occupies as much area as 5 EV5 cores. For estimating the area of EV6+, we assumed that the area overhead for adding the first extra thread to EV6 is 12% and for other threads, the overhead is 5%. These numbers were based on academic and industrial predictions [4, 12, 19].

To evaluate the performance of heterogeneous architectures, we perform comparisons against homogeneous architectures occupying equivalent area. We assume that the total area available for cores is around 100 mm^2. This area can accomodate a maximum of 4 EV6 cores or 20 EV5 cores. We expect that while a 4-EV6 homogeneous configuration would be suitable for low-TLP (thread-level parallelism) environments, the 20-EV5 configuration would be a better match for the cases where TLP is high. For studying heterogeneous architectures, we choose a configuration with 3 EV6 cores and 5 EV5 cores with the expectation that it would perform well over a wide range of available thread-level parallelism. It would also occupy roughly the same area. For our experiments with multithreading, we study the same heterogeneous configuration, except with the EV6 core replaced with an EV6+ core, and compare it with homogenous architectures of equivalent area.

For the chosen cache configuration, the area occupied by the L2 cache would be around 135 mm^2. The rest of the logic (e.g. 16 memory-controllers, crossbar interconnect etc.) might occupy up to 50 mm^2 (crossbar area calculations assume 300 bit wide links implemented in the M3/M5 layer; memory-controller area assumptions are consistent with Piranha [2] estimates). Hence, the total die-size would be approximately 285 mm^2. For studying multi-threading heterogeneous multi-core architectures in Section 4.3, as discussed above, we use a configuration consisting of 3 EV6+ cores and 5 EV5 cores. If the same estimates for L2 area and other overheads is assumed, then the die-size in that case would be around 300 mm^2. Note that actual area might be dependent on the layout and other issues, but the above assumptions provide a first-order model adequate for this study.

3.2 Workload construction

All our evaluations are done for various number of threads ranging from one through a maximum number of available processor contexts. Instead of choosing a large number of benchmarks and then evaluating each number of threads using workloads with completely unique composition, we instead choose a relatively small number of SPEC2000 benchmarks (8) and then construct workloads using these benchmarks. Table 2 summarizes the benchmarks used. These benchmarks are evenly distributed between integer benchmarks (*crafty, mcf, eon, bzip2*) and floating-point benchmarks (*applu, wupwise, art, ammp*). Also, half of them (*applu, bzip2, mcf, wupwise*) have a large memory footprint (over 175MB), while the other half (*ammp, art, crafty, eon*) have memory footprints of less than 30MB.

All the data points are generated by evaluating 8 workloads for each case and then averaging the results. A workload consisting of n threads is constructed by selecting the benchmarks using a sliding window (with wraparound) of size n and then shifting the window right by one. Since there are 8 distinct benchmarks, the window selects eight distinct workloads (except for cases when the window-size is a multiple of 8, in those cases all the selected workloads have identical composition). All of these workloads are run, ensuring that each benchmark is equally represented at every data point. This methodology for workload construction is similar to that used in [24, 18].

Diversity in server workloads is either due to different applications being run together (as in batch workloads), due to varying computational requirements of an application over time (as in media workloads), or because different threads of execution exhibit different phases, process different data streams, and are typically not in sync (as in transaction-oriented workloads). We believe that the simulated workloads are sufficiently representative of the diversity of computational requirements in a typical server workload mix.

3.3 Simulation approach

Benchmarks are simulated using SMTSIM, a cycle-accurate execution-driven simulator that simulates an out-of-order, simultaneous multithreading processor [21]. SMTSIM executes unmodified, statically linked Alpha binaries. The simulator was modified to simulate the various multi-core architectures.

The Simpoint tool [17] was used to find good representative fast-forward distances for each benchmark. Table 2 also shows the distance to which each benchmark was fast-forwarded before beginning simulation. Unless otherwise stated, all simulations involving n threads were done for $500 \times n$ million instructions. All the benchmarks are simulated using *ref* inputs.

3.4 Evaluation metrics

In a study like this, IPC (number of total instructions committed per cycle) is not a reliable metric as it would inordinately bias all the heuristics (and policies) against inherently slow-running threads. Any policy that favors high-IPC threads boosts the reported IPC by increasing the contribution from the favored threads. But this does not necessarily represent an improvement. While the IPC over a particular measurement interval might be higher, in a real system the machine would eventually have to run a workload inordinately heavy in low-IPC threads, and the artificially-generated gains would disappear. Hence, we use weighted speedup [18, 22] for our evaluations. In this paper, weighted speedup measures the arithmetic sum of the individual IPCs of the threads constituting a workload divided by their IPC on a baseline configuration when running alone. This metric makes it difficult to produce artificial speedups by simply favoring high-IPC threads.

As another axis of comparison, we also present results from open system experiments where jobs enter and leave the system at random rates. This represents a real system with variable job-arrival rates and variable service times. The systems are then compared in terms of average response time of applications as well as system queue lengths. Response time of an application, as used in this paper, is the time between job submission and job completion, and hence accounts for the queueing delays that might be incurred when the processor is busy. We believe that this is a better metric than throughput to quantify the performance of a real system with variable job inter-arrival rates and/or variable job service times.

Program	Description	fast-forward (billion instr)
ammp	Computational Chemistry	8.75
applu	Parabolic/Elliptic Partial Diff. Equations	116
art	Image Recognition/Neural Networks	15
bzip2	Compression	65
crafty	Game Playing:Chess	83
eon	Computer Visualization	55
mcf	Combinatorial Optimization	55
wupwise	Physics/Quantum Chromodynamics	88

Table 2. *Benchmarks simulated.*

4 Results

In this section, we demonstrate the performance advantage of the heterogeneous multi-core architectures for multi-threaded workloads and demonstrate job-to-core assignment mechanisms that allow the architecture to deliver on its promise. The first two subsections focus on the former, and the rest of the section demonstrates the further gains available from a good job assignment mechanism.

4.1 Static scheduling for inter-thread diversity

The heterogeneous architecture can exploit two dimensions of diversity in an application mix. The first is diversity between applications. The second is diversity over time within a single application. Prior work [17, 25] has shown that both these dimensions of diversity occur in common workloads. In this section, we attempt to separate these two effects by first looking at the performance of a static assignment of applications to cores. Note that the static assignment approach would not eliminate the need for core switching, because the best assignment of jobs to cores will change as jobs enter and exit the system.

Figure 2 shows the results comparing one heterogeneous architecture against two homogeneous architectures all requiring approximately the same area. The heterogeneous architecture that we evaluate includes 3 EV6 cores and 5 EV5 cores, while the two homogeneous architectures that we study have 4 EV6 cores or 20 EV5 cores, respectively. For each architecture, the graph shows the variation of the average weighted speedup for varying number of threads.

For the homogeneous CMP configuration, we assume a straightforward scheduling policy, where as long as a core is available, any workload can be assigned to any core. For the heterogeneous case, we use an assignment that seeks to match the optimal static configuration as closely as possible. The optimal configuration would factor in both the effect of

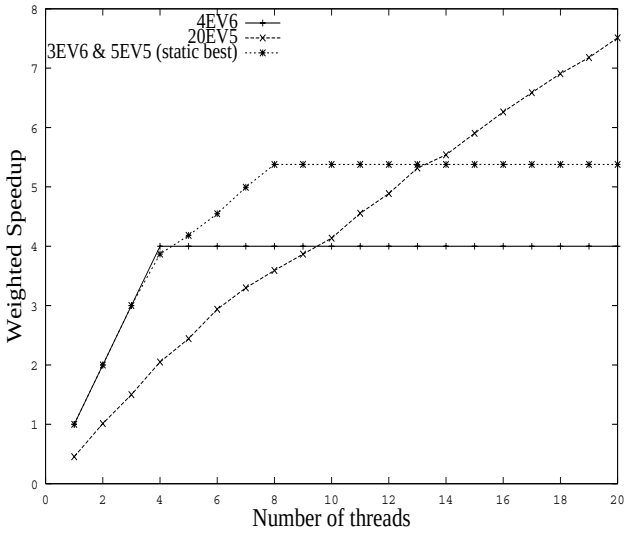

Figure 2. *Benefits from heterogeneity - static scheduling for inter-thread diversity.*

the performance difference between executing on a different core and the potential shared L2 cache interactions. However, determining this configuration is only possible by running *all possible* combinations. Instead, as a simplifying assumption, our scheduling policy assumes no knowledge of L2-interactions (only for determining core assignments – the interactions are still simulated) when determining the static assignment of workloads to cores. This simplification allows us to find the best configuration (defined as the one which maximizes weighted speedup) by simply running each job alone on each of our unique cores and using that to guide our core assignment. This results in consistently good, if not optimal, assignments. For a few cases, we compared this approach to an exhaustive exploration of all combinations; our results indicated that this results in performance close to the optimal assignments.

The use of weighted speedup as the metric ensures that those jobs assigned to the EV5 are those that are least affected (in relative IPC) by the difference between EV6 and EV5. In both the homogeneous and heterogeneous cases, once all the contexts of a processor get used, we just assume that the weighted speedup will level out as shown in the Figure 2. The effects when the number of jobs exceeds the number of cores in the system (e.g., additional context switching) is modeled more exactly in Section 4.2.

As can be seen from Figure 2, even with a simple static approach, the results show a strong advantage for heterogeneity over the homogeneous designs, for most levels of threading. The heterogeneous architecture attempts to combine the strengths of both the homogeneous configurations - CMPs of a few powerful processors (EV6 CMP) and CMPs of many less powerful processors (EV5 CMP). While for low threading levels, the applications can run on powerful EV6 cores resulting in high single thread performance, for higher threading levels, the applications can run on the added EV5 contexts enabled by heterogeneity, resulting in higher overall throughput.

The results in Figure 2 show that the heterogeneous configuration achieves performance identical to the homogeneous EV6 CMP from 1 to 3 threads. At 4 threads, the optimum point for the EV6 CMP, that configuration shows a slight advantage over the heterogeneous case. However, this advantage is very small because with 4 threads, the heterogeneous configuration is nearly always able to find one thread that is impacted little by having to run on an EV5 instead of EV6. As soon as we have more than 4 threads, however, the heterogeneous processor shows clear advantage.

The superior performance of the heterogeneous architecture is directly attributable to the diversity of the workload mix. For example, *mcf* underutilizes the EV6 pipeline due to its poor memory behavior. On the other hand, benchmarks like *crafty* and *applu* have much higher EV6 utilization. Static scheduling on heterogeneous architectures enables the mapping of these benchmarks to the cores in such a way that overall processor utilization (average of individual core utilization values) is maximized.

The heterogeneous design remains superior to the EV5 CMP out to 13 threads, well beyond the point where the heterogeneous architecture runs out of processors and is forced to queue jobs. Beyond that, the raw throughput of the homogeneous design with 20 EV5 cores wins out. This is primarily because of the particular heterogeneous designs we chose. However, more extensive exploration of the design space than we show here confirms that we can always come up with a different configuration that is competitive with more threads (e.g., fewer EV6's, more EV5's), if that is the desired design point.

Compared to a homogeneous processor with 4 EV6 cores, the heterogeneous processor performs up to 37% better with an average 26% improvement over the configurations considering 1-20 threads. Relative to 20 EV5 cores, it performs up to 2.3 times better, and averages 23% better over that same range.

These results demonstrate that over a range of threading levels, a heterogeneous architecture can outperform comparable homogeneous architectures. Although the results are shown here only for a particular area and two core types (as discussed in Section 3.1), our experiments with other configurations (at different processor areas and core types) indicate that these results are representative of other heterogeneous configurations as well.

4.2 Open system experiments

Graphs of performance at various threading levels are instructive, but do not necessarily reflect accurately real system performance. Real systems typically operate at a variety of threading levels (run queue sizes), and observed performance is a factor of the whole range. Thus, while a particular architecture may appear to be optimal at a single point (even if

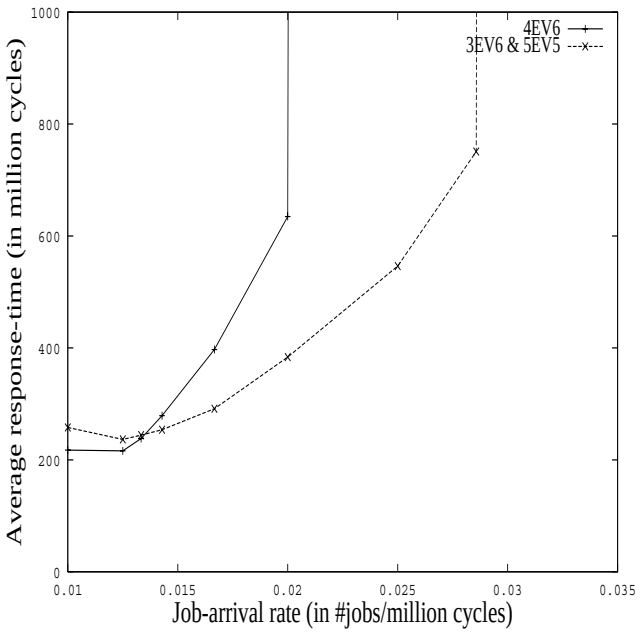

Figure 3. *Limiting response-time for various loads.*

that design point represents the expected average behavior), it may never be optimal on a system that experiences a range of demand levels. This section explores the performance of heterogeneous architectures on an open system. It does so with a sophisticated simulation framework that models random job arrivals and random job lengths. This addresses some methodological issues that remain, even when using the weighted speedup metric. In this experiment, we are able to guarantee that every simulation executes the exact same set of instructions. Additionally, we are able to use average response time as our performance metric.

We model a system where jobs enter and leave the system with exponentially distributed arrival rate λ and exponentially distributed average time to complete a job T. We study the two systems for varying values of λ and observe the effects on mean response time, queue length, and stability of the systems. Whenever a system is stable, it is better to measure response time rather than throughput, since throughput cannot possibly exceed the rate of job arrival. If two stable systems are compared and one is faster, the faster one will complete jobs more quickly and thus typically have fewer jobs queued up waiting to run.

For these experiments, we randomly generate jobs (using a Poisson model) centered around an average expected execution time of 200 million cycles on an EV6. Jobs are generated by first generating random numbers with average distribution centered around 200 million cycles and then executing that many instructions multiplied by the single-threaded IPC of the benchmarks on EV6. We then simulate different mean job arrival rates with exponential distributions. To model a random system but produce repeatable results, for each point on the job arrival rate axis, we feed the same jobs in the same order with the same arrival times to each of the systems.

For the heterogeneous configuration, we use a naive scheduling heuristic which simply assigns jobs randomly, only ensuring that the more powerful processors get used before the less powerful. Significant improvements over this heuristic will be demonstrated in the following sections.

Figure 3 shows the results for these experiments. The most profound difference between the homogeneous and the heterogeneous architectures is that they saturate at very different throughputs. The homogeneous architecture sees unbounded response times as the arrival rate approaches its maximum throughput around 2 jobs per 100 million cycles. At this point, its run queue becomes (if we ran the simulations long enough) infinite.

However, the heterogeneous architecture remains stable well beyond this point. Furthermore, the scheduling heuristics we will demonstrate in the following section will actually increase the maximum throughput of the architecture, so its saturation point would be even further out. The heterogeneous architecture also sees average response time improvements well before the other architecture becomes saturated. There is only a very narrow region where the homogeneous architecture sees no queueing beyond 4 jobs, but the heterogeneous is forced to use an EV5 occasionally, where the homogeneous architecture sees some slight advantage. As soon as the probability of queue lengths beyond four becomes non-insignificant, the heterogeneous architecture is superior.

Another interesting point to note is that, besides supporting greater throughput in peak load conditions, heterogeneous chip-level multiprocessor response time degrades more gracefully under heavier loads than for homogeneous processors. This should enhance system reliability in transient high load conditions. This is particularly important as reliability and availability of systems become more important with the maturity of computer technology.

4.3 Dynamic scheduling for intra-thread diversity

The previous sections demonstrated the performance advantages of the heterogeneous architecture when exploiting core diversity for inter-workload variation. However, that analysis has two weaknesses – it used unimplementable assignment policies in some cases (e.g., the static assignment oracle) and ignored variations in the resource demands of individual applications. This section solves each of these problems, and demonstrates the importance of good dynamic job assignment policies.

Prior work has shown that an application's demand for processor resources varies across phases of the application. Thus, the best match of applications to cores will change as those applications transition between phases. In this section, we examine implementable heuristics that dynamically adjust the mapping to improve performance.

These heuristics are sampling-based. During the execution of a workload, every so often, a trigger is generated that initiates a *sampling phase*. In the *sampling* phase, the scheduler

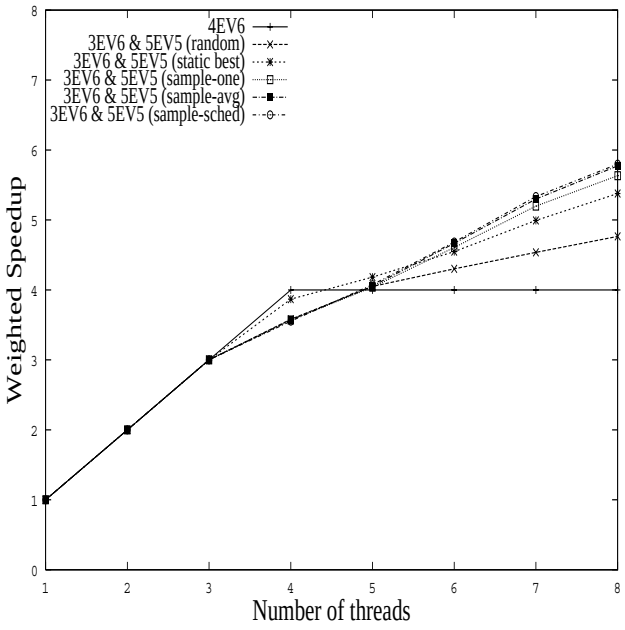

Figure 4. *Three strategies for evaluating the performance an application will realize on a different core.*

permutes the assignment of applications to cores, changing the cores onto which the applications are assigned. During this phase, the dynamic execution profiles of the applications being run are gathered by referencing hardware performance counters. These profiles are then used to create a new assignment, which is then employed during a much longer phase of execution, the *steady* phase. The steady phase continues until the next trigger. Note that applications continue to make forward progress during the sampling phase, albeit perhaps non-optimally.

4.3.1 Core sampling strategies

There are a large number of application-to-core assignment permutations possible, both for the sampling phase and for the steady phase. We prune the number of permutations significantly by assuming that we would never run an application on a less powerful core when doing so would leave a more powerful core idle (for either the sampling phase or the steady phase). Thus, with four threads on our 3 EV6/5 EV5 configuration, four possible assignments are possible based on which thread gets allocated to the EV5. With more threads, the number of permutations increase, up to 56 potential choices with eight threads. Rather than evaluating all these possible alternatives, our heuristics only sample a subset of possible assignments. Each of these assignments are run for 2 million cycles. At the end of the sampling phase, we use the collected data to make assignments.

Selection of the assignments to be sampled depends on how much we account for interactions at the L2 cache level (which, if large, can color the data collected for all threads and lead to inappropriate decisions). We evaluated three strategies for sampling the assignment space.

The first strategy, *sample-one*, samples as many assignments as is needed to run each thread once on each core-type. This assumes that the single sample is accurate, regardless of what other jobs are doing. Then the assignment is made, maximizing weighted speedup under the assumption future performance will be the same as our one sample for each thread. The assignment that maximizes weighted speedup is simply the one that assigns to the EV5s those jobs whose ratio of average EV5 throughput to EV6 throughput is highest.

The second strategy, *sample-avg*, assumes we need multiple samples to get the average behavior of a job on each core. In this case, we sample as many times as there are threads running. The samples are distinct and are done such that we get at least two runs of each thread on each core type, then base the assignment (again maximizing expected weighted speedup) on the average performance of each thread on each core.

The third strategy, *sample-sched*, assumes we know little about a particular assignment unless we have actually run it. It thus samples a number of possible assignments, and then is constrained to choose one of the assignments it sampled. In fact, we sample $4 \times n$ representative assignments for a n-threaded workload (bounded by the maximum allowed for that configuration). Selection of the best core assignment, of those sampled, is the one that maximizes total weighted speedup, using average EV5 throughput for each thread as the baseline.

Figure 4 presents a quantitative comparison of the effectiveness of the three strategies. The average weighted speedup values reported here were obtained using a default time-based trigger that resulted in a sampling phase being triggered every 500 million processor cycles; Section 4.3.2 evaluates the impact of other time intervals. Also included in the graph, for comparison, are (1) the results obtained using the homogeneous multi-core processor, (2) the random assignment policy described in the previous section, and (3) the best static assignment found previously.

As suggested by the graph, the *sample-sched* strategy performs the best, although *sample-avg* has very similar performance (within 2%). Even *sample-one* is not much worse. We observed a similar trend for other time intervals and for other trigger types. We conclude from this result that for our workload and L2 cache configuration, the level of interaction at the L2 cache is not sufficient to affect overall performance unduly. We use *sample-avg* as the basis for our trigger evaluation in the sections to follow as it not only has lower overhead than *sample-sched*, but is also more robust than both *sample-sched* and *sample-one* against worst-case events like phase changes *during* sampling.

The second significant result that the graph shows is that the intelligent assignment policies make a significant performance difference, allowing us to outperform the random core assignment strategy by up to 22%. Perhaps more surprising

is the importance of the dynamic sampling and reconfiguration, as we outperform the static best by as much as 10%. We take a hit at 4 threads, when the sampling overhead first kicks in, but quickly recover that loss as the number of threads increases, maximizing the scheduler's flexibility. The results also suggest that fairly good decisions can be made about an application's relative performance on various cores even if it runs for no more than 2 million cycles on each core. This also indicates that the cold-start effect on core-switching is *much less* than the running time on a core during sampling. More discussion about sampling overhead can be found in the next section.

4.3.2 Trigger mechanisms

Sampling effectively requires juggling two conflicting goals – minimizing sampling overhead and reacting quickly to changes in workload behavior. To manage this tradeoff, we compare two classes of trigger mechanisms, one based on a periodic timer, and the second based on events indicating significant changes in performance.

We begin by evaluating the first class and the performance impact of varying the amount of time between sampling phases, that is, the length of the steady phase. For smaller steady phase lengths, a greater fraction of the total time is spent in the sampling phases, thus contributing overhead. The overhead derives from, first, the overhead of application core switching each time we sample a different configuration, and second, the fact that sampling by definition is usually running non-ideal configurations.

Figure 5 presents a comparison of the average weighted speedup obtained with steady-phase lengths between 31.25 million and 500 million cycles for the sample-average strategy. We note from this graph that the sampling frequency has a second-order impact on performance while the steady-phase length of 125 million cycles performs best overall. Also, in a similar observation as in [14], we found that the act of core-switching has relatively small overhead. So, the optimal sampling frequency is determined by the average phase length for the applications constituting the various workloads, as well as the ratio of the lengths of the steady phase and the sampling phase.

While time-triggered sampling is very simple to implement, it does not capture either inter-thread or intra-thread diversity fully. A fixed sampling frequency is inadequate when the phase lengths of different applications in the workload mix are different. Also, each application can demonstrate multiple phases each with its own phase length. For example, in our simulation window, while *art* demonstrates a periodic behavior with phase length of 80 million instructions, *mcf* demonstrates at least two distinct phases with one of the phases at least 350 million instructions long. Any sampling-based heuristic that hopes to capture phase changes for both *art* and *mcf*, with minimal overhead, needs to be adaptive.

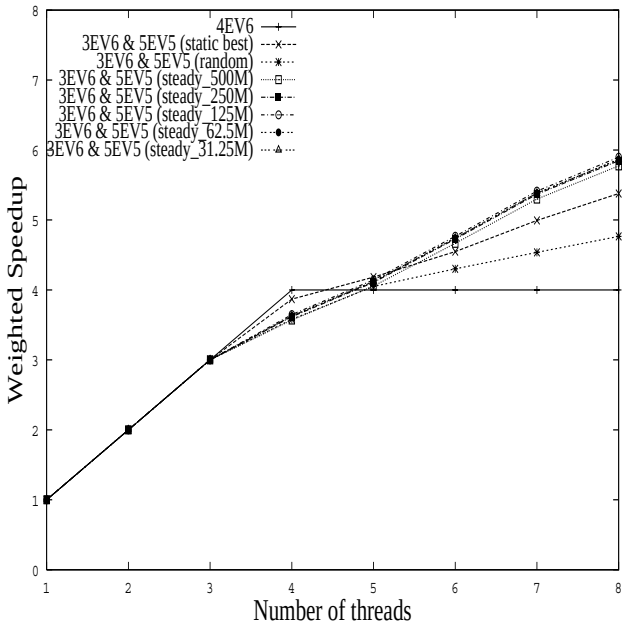

Figure 5. *Sensitivity to sampling frequency for time-based trigger mechanisms using the sample-avg core-sampling strategy*

Next, we consider the second class of trigger mechanisms. Here, we monitor the run-time behavior of the workload and detect when sufficiently significant changes have occurred. We consider three instantiations of this trigger class. With the *individual-event* trigger, a sampling phase is triggered every time the steady-phase IPC of an individual thread changes by more than 50%. In contrast, with the *global-event* trigger, we sum the absolute values of the percent changes in IPC for each application, and trigger a sampling phase when this value exceeds 100%. The last heuristic, *bounded-global-event*, modifies the *global-event* trigger by initiating a sampling phase if more than 300 million cycles has elapsed since the last sampling phase, and avoiding sampling if the global event trigger occurs within 50 million cycles since the last sampling phase. All the thresholds were determined by observing the execution characteristics of the simulated applications.

Figure 6 presents a comparison of these three event-based triggers, along with the time-based trigger mechanism using a steady-state length of 125 million cycles (the best one from the previous discussion). We continue to use the sample-avg core-sampling strategy. The graph also includes the static-best heuristic from Section 4.1, and the homogeneous core. As we can see from the figure, the event-based triggers outperform the best timer-based trigger and the static assignment approach. This mechanism effectively meets our two goals of reacting quickly to workload changes and minimizing sampling.

While the individual event trigger performs well in general, using a global event trigger achieves better performance. This

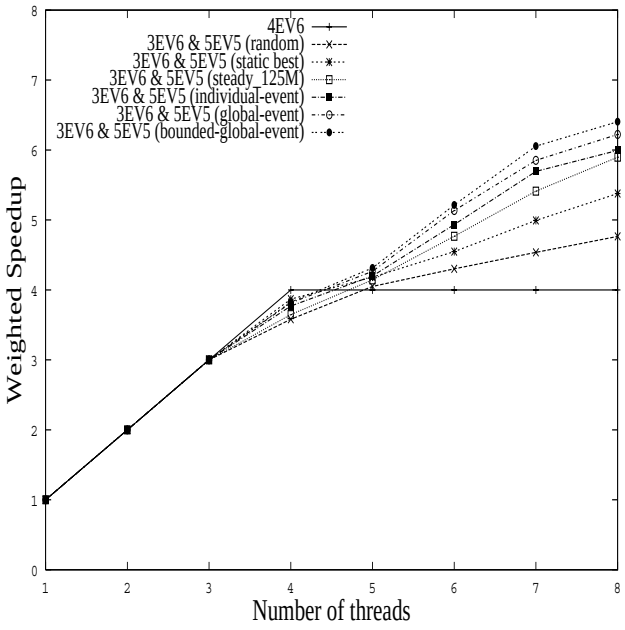

Figure 6. *Comparison of event-based triggers using the sample-avg core-sampling strategy.*

is because a change in the behavior of a single application might often not result in changing the workload-to-cores mapping. Using a global event trigger guards against these false positives. The bounded-global-event trigger achieves the best performance (close to a 20% performance improvement over static) indicating the benefits from a hybrid timer-based and event-based approach. It has all the advantages of a global-event trigger, but the bounds also help to guard against false positives and false negatives. By eliminating most of the sampling overhead, this hybrid scheme also closes the gap again with the homogeneous processor at 4 threads. Similar trends were observed for different values of parameters embodied in the event-based triggers.

4.3.3 Summary

The results presented in this section indicate that dynamic heuristics which intelligently adapt the assignment of applications to cores can better leverage the diversity advantages of a heterogeneous architecture. Compared to the base homogeneous architecture, the best dynamic heuristic achieves close to a 63% improvement in throughput in the best case (for 8 threads) and an average improvement in throughput of 17% over configurations running 1-8 threads. Even more interesting, the best dynamic heuristic achieves a weighted speedup of 6.5 for eight threads, which is close to 80% of the optimal speedup (8) achievable for this configuration (despite the fact that over half of our cores have roughly half the raw computation power of the baseline core!). In contrast, the homogeneous configuration achieves only 50% of the optimal speedup. We have also demonstrated the importance of the intelligent dynamic assignment, which achieves up to 31% improvement over a random scheduler.

While our relatively simple dynamic heuristics are effective, there are clearly many other heuristics that could be considered. For example, event-based sampling could be based on other metrics aside from IPC, such as changes in ILP, cache or branch behavior, or basic block profiles as suggested in [17]. Further, rather than using past behavior as a simple approximation for future behavior, more complex models for phase identification and prediction [17] could also be used.

4.4 Cores supporting multithreading

The availability of multithreaded cores adds one more dimension of heterogeneity to exploit. While the previous section demonstrated that interactions between scheduled threads could be largely ignored when that interaction occurs at the L2 cache, jobs co-scheduled on a simultaneous multithreading processor interact at a completely different level, sharing and competing for virtually all processor resources. In this case, interactions cannot be ignored. For example, running *ammp* co-scheduled with *mcf* tells us little about the performance of *ammp* co-scheduled with *bzip*. Prior research has indeed shown that not all combinations of jobs coexist with the same efficiency on a multithreaded processor [18], a phenomenon called *symbiosis*.

In a heterogeneous design with only some cores multithreaded, those jobs that do have high symbiosis will migrate to the multithreaded cores (assuming intelligent core assignment), resulting in more efficient multithreaded execution than if all cores were run in multithreaded mode.

However, the multithreaded processor core creates a significant challenge to the core scheduler. First, because jobs interact at such an intimate level on an SMT processor, the simpler sampling policies that ignored job interactions will not work well, meaning we should only consider scheduling the permutations we are willing to sample. Second, the permutation space of potential assignments is itself much larger. Consider just one multithreaded core and several smaller cores. With 4 jobs, if the larger core were not multithreaded, we would consider only four permutations. But with the multithreaded core, it is not clear whether one, two, three, or four jobs should go on the faster core before we decide to use the others. In fact, there are now 15 permutations that have reasonable chances of being the best. It should be noted that this problem is not unique to heterogeneous cores. A homogeneous SMT multiprocessor will require very similar sampling just to find the optimal partitioning of jobs to cores. In fact, the application of our heterogeneous scheduling principles to a homogeneous SMT CMP architecture, and a comparison of the two architectures, is the subject of future research.

For a system with more multithreaded cores and more threads, the sample space becomes unmanageable very quickly. To test some scheduling mechanisms for a heterogeneous architecture which includes multithreaded cores, we

change our base heterogeneous configuration to have the EV6 processors modified to support multithreading (which we call EV6+) while continuing to have 5 EV5 cores as before. This uses 4.66 times the area of an EV6 core and 22.6 times the area of an EV5 core. We chose EV6+ for this study over other possible multithreaded configurations, such as EV8, because in the time-frame we are targeting multiple EV8 cores per die are not as realistic. Also, we wanted results comparable with the rest of this paper. Multithreading is less effective on EV6s, but the area-performance tradeoffs are much better than the large EV8.

Because EV6+ is a narrower width machine than assumed in most prior SMT research, we expect the incremental performance of the 2nd, 3rd, and 4th thread to be lessened. For this reason, we will make the following assumption which helps prune the sample space considerably – we assume that it is always better to run on an EV5 than to be the third thread on an EV6+. Even with this assumption, we still need sampling to know whether a thread would be better off as the 2nd thread on an EV6+ or alone on the EV5.

For these remaining choices, we examine the following sampling strategies for this architecture. *pref-EV6* always assumes it is best to run 2 threads on each EV6+ before using the EV5s, and samples accordingly. *pref-EV5* always assumes it is best to run on the EV5 rather than put two threads on an EV6+ and samples accordingly. *pref-neither* assumes either is equally likely to be optimal and samples random schedules within the constraint that we don't put a third thread on a core until all EV5's are used. *pref-similar* assumes that the configuration we are running with now has merit, and that the next configuration will be similar. In this case, sampling is biased toward a similar configuration (same number of total jobs on the EV6+'s), with only about 30% of the samples deviating from that.

For all these heuristics, the sampling phase is again 2 million cycles while the steady phase is 500 million cycles. Every experiment involving n threads involves $2 \times n$ samples. Performance of *pref-similar* does depend on the initial configuration. We started each simulation with the jobs in a known good assignment to better reflect steady state behavior.

The results with multithreaded cores are shown in Figure 7. Also included in the graph are random scheduling results as well as an extrapolation of the homogeneous results from earlier graphs. Those results are extrapolated to a hypothetical 4.66 EV6 (*eqv-EV6-homogeneous*) and a hypothetical 22.6 EV5 (*eqv-EV5-homogeneous*) configuration, to provide more accurate/fair comparisons at constant area. These results show that the core assignment policy is indeed more important to this architecture than the non-multithreaded cores. Compared to the differences between the *sample-one*, *sample-avg*, and *sample-sched* (Figure 4), the differences between the heuristics in Figure 7 are significantly higher. The trends here are also similar to those in the earlier section that showed that the more permutations available to the scheduler, the more opportunity there was to choose a good assignment schedule. It also shows that with fewer threads available it is best to be somewhat pessimistic about the performance of additional threads on the EV6+. Of course, this is an artifact of the particular multithreaded architecture we model, rather than a general result. The adaptive *pref-similar* technique, which samples the immediate neighborhood of the current configuration provides significant benefit over the entire range. This indicates that (1) there is typically not a clear answer between whether to put extra threads on EV6 or to use another EV5, (2) there is significant value in making the right decision in this respect, and (3) using the current configuration as a pruning tool is effective in finding a good assignment.

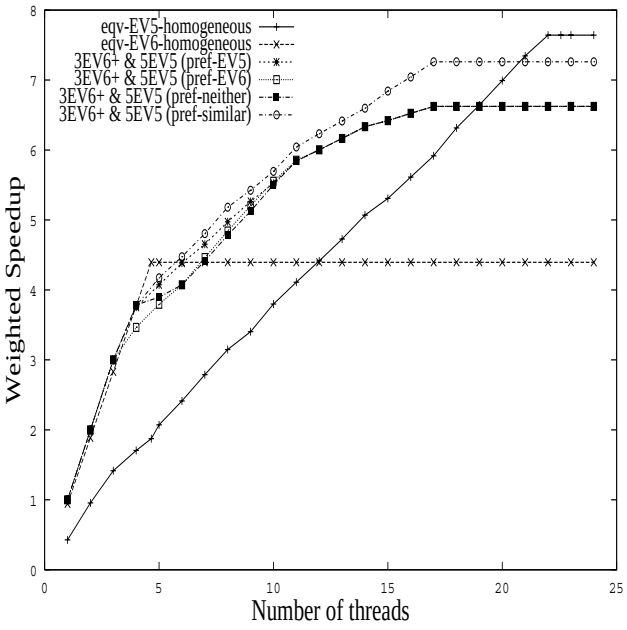

Figure 7. *Performance of heuristics for a heterogeneous architecture with multithreaded cores.*

With the addition of modest multithreading cores to our heterogeneous architecture, we provide much better performance with higher threading levels. In fact, we significantly reduce the portion of the graph where many smaller processors are optimal, relative to Figure 2.

5 Conclusions

We have demonstrated that heterogeneous multi-core architectures can provide significant throughput advantages over equivalent-area homogeneous multi-core architectures. This throughput advantage results from the ability of heterogeneous processors to better exploit both variations in thread-level parallelism as well as inter- and intra- thread diversity. We also propose and evaluate a set of thread scheduling mechanisms to best realize the potential performance gain available from heterogeneity.

Over a wide range of threading parallelism, the representative heterogeneous architecture we study perform 18% better

on average than a homogeneous CMP architecture of the same area on SPEC workloads. For an open system with random task arrivals and exits, our results showed that heterogeneous architectures can have much lower response times than corresponding homogeneous configurations. Also, the heterogeneous systems were stable at job arrival rates that were up to 43% higher.

Having a diversity of cores with varying resources and pipeline architectures enables the system to efficiently leverage application diversity both at the inter-thread and intra-thread level. Applications least able to derive significant benefits from large and complex cores can instead be run on smaller, less complex cores with much better area efficiencies.

This work demonstrates effective yet relatively simple task scheduling mechanisms to best match the applications to cores. Our best core assignment strategy achieves more than a 30% performance improvement over a naive heuristic, while still being straightforward to implement. Relatively simple heuristics are also demonstrated to be effective even when one or more of the heterogeneous cores are multithreaded.

Acknowledgments

The authors would like to thank the reviewers for helpful feedback, Jeff Brown for significant help with the simulation tools, and Brad Calder for helpful discussions. This research was funded in part by NSF grant CCR-0105743 and a grant from Intel Corporation.

References

[1] International Technology Roadmap for Semiconductors 2003, http://public.itrs.net.

[2] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[3] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In *Proceedings of the IEEE International Conference on Cluster Computing*, Oct. 2001.

[4] J. Burns and J.-L. Gaudiot. SMT layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(2), Feb. 2002.

[5] R. H. Denard. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-state Circuits*, 98(5), 1974.

[6] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A vector extension to the alpha architecture. In *International Symposium on Computer Architecture*, May 2002.

[7] R. Figueiredo and J. Fortes. Impact of heterogeneity on DSM performance. In *Sixth International Symposium on High-Performance Computer Architecture*, Jan. 2000.

[8] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997.

[9] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2002.

[10] J. L. Hennessy and N. P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, Sept. 1991.

[11] IBM. Power4:http://www.research.ibm.com/power4.

[12] IBM. Power5: Presentation at microprocessor forum. 2003.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Processor power reduction via single-ISA heterogeneous multi-core architectures. In *Computer Architecture Letters, Vol 2*, Apr. 2003.

[14] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, Dec. 2003.

[15] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[16] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par, Vol. II*, 1996.

[17] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. In *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, Dec. 2003.

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

[19] Sun. UltrasparcIV: http://siliconvalley.internet.com/news/print.php/3090801.

[20] H. Topcuoglu, S. Hariri, and M. Wu. Task scheduling algorithms for heterogeneous processors. In *8th IEEE Workshop on Heterogeneous Computing Systems*, Apr. 1999.

[21] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[22] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[23] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[24] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[25] D. Wall. Limits of instruction-level parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.