Journal of Instruction-Level Parallelism 10 (2008) 1-26 Submitted 10/06; published 6/08

Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures

Michela Becchi MBECCHI@CSE.WUSTL.EDU
Patrick Crowley PCROWLEY@WUSTL.EDU
Department of Computer Science and Engineering
Applied Research Laboratory
Washington University
One Brookings Drive
St. Louis, MO 63130-4899 USA

Abstract

In a multi-programmed computing environment, threads of execution exhibit different runtime characteristics and hardware resource requirements. Not only do the behaviors of distinct threads differ, but each thread may also present diversity in its performance and resource usage over time. A heterogeneous chip multiprocessor (CMP) architecture consists of processor cores and caches of varying size and complexity. Prior work has shown that heterogeneous CMPs can meet the needs of a multi-programmed computing environment better than a homogeneous CMP system. In fact, the use of a combination of cores with different caches and instruction issue widths better accommodates threads with different computational requirements.

A central issue in the design and use of heterogeneous systems is to determine an assignment of tasks to processors which better exploits the hardware resources in order to improve performance. In this paper we argue that the benefits of heterogeneous CMPs are bolstered by the use of a dynamic assignment policy, i.e., a runtime mechanism which observes the behavior of the running threads and exploits thread migration between cores. We validate our analysis by means of simulation. Specifically, our model assumes a combination of Alpha EV5 and Alpha EV6 processors and of integer and floating point programs from the SPEC2000 benchmark suite. We show that a dynamic assignment can outperform a static one by 20% to 40% on average and by as much as 80% in extreme cases, depending on the degree of multithreading simulated.

1. Introduction

Chip multiprocessors (CMPs) will dominate commercial processor designs for at least the next decade, during which we will likely see an annual doubling of the number of processor cores integrated onto a single chip. This development is driven by technological constraints. It is possible to efficiently scale a CMP design by increasing the number of cores while maintaining or reducing overall power consumption by reducing clock frequency. As long as the increase in cores offsets the reduction in clock frequency, peak system performance will improve.

While replicating cores is an efficient strategy, architects are confronted with a basic question: what type of core should be replicated? A given die area can accommodate: many small, simple cores; fewer cores of a larger more sophisticated variety; or some combination of the two. Thus, in CMPs it is common to see either many simple processors [14] or a moderate quantity of high performance cores [13]. The first solution meets the needs of computing environments characterized by higher thread parallelism, while the second better accommodates scenarios with lower thread parallelism and higher individual thread complexity.

Heterogeneous CMP systems, consisting of a combination of processor cores of varying type on the same chip, have been proposed recently as a compromise between the two alternatives above. The idea comes from the observation that a multi-programmed computing environment may present threads of execution with different hardware resource requirements, and that such needs may vary over time. Hence, an appropriate mapping of different threads to heterogeneous processor cores can maximize resource utilization and, at the same time, achieve a high degree of inter-thread parallelism.

In order to take advantage of a heterogeneous architecture, an appropriate policy to map running tasks to processor cores must be determined. The overall goal of such a strategy must be to maximize the performance of the whole system by accurately exploiting its resources. The control mechanism must take into account the heterogeneity of the system and of the workload, and the varying behavior of the threads over time. Moreover, it must be easily implementable and introduce as little overhead as possible.

In this paper we argue that, in a heterogeneous CMP system with a multi-programmed workload, a dynamic policy, i.e. a mechanism which observes the runtime behaviors of the running threads and exploits thread migration between the cores, is preferable to a static assignment. Our evaluation takes into consideration various combinations, both homogeneous and heterogeneous, of Alpha EV5 and Alpha EV6 processors all occupying the same die area. The workloads consist of several combinations of programs from the SPEC2000 benchmark suite.

We compare two homogeneous and three heterogeneous CMP configurations when a static assignment policy (not involving thread migration) is used. Specifically, we distinguish an average case from an ideal case, which assumes a priori knowledge of the performance of each thread on the two kinds of processors in use. We finally define two dynamic assignment policies, round robin and IPC-driven, and compare their performance with the static case.

We show that a heterogeneous system adopting a dynamic assignment policy is able to accommodate a variety of degrees of thread parallelism more efficiently than both a homogeneous and a heterogeneous system adopting a static assignment policy, and we quantify the performance improvements over both of them.

The rest of the paper is organized as follows. In Section 2 we introduce the problems addressed in this paper through a simple example. In Section 3 we describe our simulation methodology, processor configurations, and workload configuration. In Section 4 we present an analysis of the behavior of the adopted benchmarks on the two considered Alpha cores. In Section 5 we describe our simulation model. In section 6 we detail describe the static and dynamic thread assignment policies. In Section 7 we present the results of simulations performed on homogeneous and heterogeneous CMP architectures. In Section 8 we present an analytical model allowing us to generalize our results and extend them to different processor types. In Section 9 we briefly relate our work to the literature. Finally, in Section 10 we summarize the goals and the results of our analysis.

2. Background

In this section, we develop a simple example highlighting the importance of dynamic thread scheduling in a heterogeneous CMP system, while also clarifying our target architecture and workloads.

	P1	**P2**
Thread A	1.6	0.4
Thread B	1.5	1

Table 1: IPC of threads A and B on cores P1 and P2.

Our interest is in heterogeneous CMP systems executing throughput-oriented workloads. In a throughput-oriented system, we are concerned with the total execution time for a group of programs, rather than the execution time of any individual one.

Each program in such a group will likely exhibit a different level of performance on each processor core type in the heterogeneous CMP. That is, a given program may execute a different number of instructions per cycle (IPC) on each core type due to the varying microarchitectural resources and capabilities present (e.g., caches and branch predictors).

Consider a scenario with the simplest heterogeneous CMP system, consisting of two processors P1 and P2 of different type, and a workload of two programs, thread A and thread B. Suppose further that each program exhibits the average IPC values reported in Table 1. From this data, we can infer that P1 is a more sophisticated, higher performance processor than P2. Thus, thread A experiences a four-fold IPC improvement on P1 relative to P2, while thread B sees an improvement of 50% in average IPC. To further simplify the example (we will tackle a more realistic scenario later), we assume that each program will run for 1 million instructions.

We now consider possible mappings of these programs onto the cores. Given our simple assumptions, we can calculate the execution time for each program on each processor core by dividing the total number of instructions (again, assumed to be 1 million) by the average IPC on the target processor. For example, if we map thread A onto P2 and thread B onto P1, thread A will complete in 2.5M cycles whereas thread B will complete in around 700,000 cycles. The total execution time, then, is the maximum of these two, 2.5M cycles. Alternately, if we map thread A to P1 and thread B to P2, thread B becomes the longer running program leading to a total execution time of 1M cycles. This introductory example illustrates a 2.5X difference in performance due to the mapping of programs to cores. These results are shown in Figure 1.

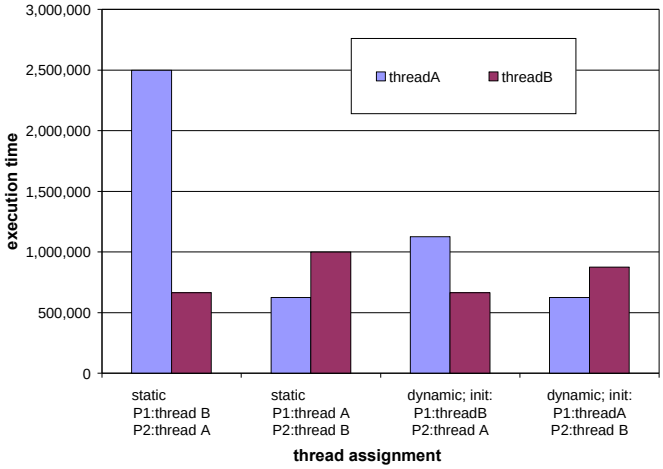

Figure 1: Execution times with different mapping of two threads onto heterogeneous dual core system.

The two assignments above represent static assignments, that is, the thread-to-processor mapping does not change once made. If we assume that programs can migrate across cores, however, we can explore dynamic assignment policies. Suppose, for example, that our system always keeps P1 busy. That is, whenever P1 becomes idle the program running on P2 is moved immediately to P1. As shown in Figure 1, in the example this policy improves on the best static mapping by 20%. As we will see, the difference is considerably greater in more realistic scenarios.

Note that the importance of thread assignment depends on the ratios between the IPCs on the two different core types. Specifically, the higher the ratios, the more the execution time will depend on the mapping performed and the dynamic assignment will outperform the static one. This can also be intuitively understood thinking that unitary ratios correspond to a homogeneous system, where remapping does not find a reason to exist.

In the remainder of this paper we expand the basic ideas introduced in this example and address some open questions.

First, in our example above, each program had sufficiently different performance on each core type to demonstrate the importance of thread assignment. But do real programs running on real processor cores have such performance differentials? In Section 4, we show that SPEC programs executing on two Alpha core types do.

Second, we remove most of the simplistic assumptions in the model above and study more realistic scenarios. In particular, our analysis will take into consideration: (i) a higher number of cores and programs, (ii) the fact that IPC is not known a priori, (iii) the fact that IPC is not a constant parameter but varies over time during program execution, and (iv) thread migration overhead. We will show how this added complexity further motivates the use of dynamic thread assignment.

Third, by gradually introducing assignment policies with increasing complexity we point out the reasons and the conditions which make dynamic assignment suitable in a fully heterogeneous environment.

3. Architecture and Methodology

In this section we describe our experimental methodology, the processor configuration for both homogeneous and heterogeneous systems, the workload setup and the evaluation metrics.

3.1. Simulation Approach

In our evaluation we consider homogeneous and heterogeneous configurations of EV5 (Alpha 21164) [5] and EV6 (Alpha 21264) [15] processors. Since it would be very time consuming to perform full system simulations for each combination of CMP configuration, workload setup, assignment policy, our experiments are conducted in two phases.

We first use the M5 simulator [6] to gather execution traces for each program on both core types. We then use our own simulator to model each CMP configuration and to evaluate different assignment policies.

The traces collected in the first phase are used to model the execution of each thread in the CMP simulations performed in the second phase. Each trace entry corresponds to the average IPC computed on a window of 1M clock cycles. Hence, the execution of the CMP system is modeled by decomposing it in windows of 1M clock cycles, and advancing each thread according to the

	EV5 (Alpha 21164)	EV6 (Alpha 21264)
Issue-width	4-issue	6-issue
L1 caches	8KB/DM/32B blocks	64KB/2-way/64B blocks
# MSHRs	4	8
Branch pred	2K-gshare	Hybrid
Pipelines	2 INT, 2 FP	4 INT, 2 FP
Area	5.06 mm2	24.05 mm2
L2 cache	unified, 4MB, 128 B blocks	
Latencies	L1 2 clock cycles, L2 10 clock cycles, main memory 150 ns	
Bus L2-main memory	2GB/s bandwidth	
Clock	2.1 GHz	

Table 2: Hardware characteristics of EV5 and EV6 cores.

appropriate IPC value. Note how using traces is more realistic and removes the assumption of constant IPC made in the introductory example in the background section.

This two-phase approach has the benefit of flexibility and fast execution, and the drawback of neglecting the effects of L2 cache contentions. However, we evaluated the impact of L2 cache conflicts for a subset of the results by reducing L2 cache size to 1/Nth of the original size (for N cores) and saw no major effects. A detailed discussion of our simulation model is given in Section 5.

3.2. Processor Configurations

We have developed M5 simulator configurations to represent EV5 and EV6 processors. The basic aspects of the hardware setup are detailed in Table 2.

Notice that EV6 offers higher ILP and bigger and more sophisticated L1 caches at the cost of higher area occupancy. Also, note that the area needed for one EV6 can accommodate approximately 5 EV5s. Therefore, in order to consider a constant and equal amount of area, in our experiments we make use of homogeneous configurations consisting of 4 EV6s or 20 EV5s and the following heterogeneous configurations: 5 EV5s and 3 EV6s, 10 EV5s and 2 EV6s, 15 EV5s and 1 EV6.

3.3. Workload Definition

Our workloads are based on eleven programs from the SPEC2000 benchmark suite; five of them are integer (*gcc*, *gzip*, *bzip2*, *parser* and *crafty*) and six floating point (*equake*, *galgel*, *lucas*, *wupwise*, *mgrid* and *swim*). All these benchmarks are run in M5 using the *ref* SPEC input option. The programs are summarized in Table 3.

In our CMP simulations, we vary the number of running threads from one to forty (twice the maximum amount of processors used). Both EV5 and EV6 are single threaded processors: in situations where the number of running programs exceeds the one of available cores, some queuing results.

In order to reduce the sensitivity of the results to the particular set of programs simulated, workloads are constructed with randomization. Specifically, for a given number of threads and for each distinct CMP configuration, 100 different simulations are run; we report the averages. In

Program	Description
164.gzip	Data compression utility
176.gcc	C compiler
186.crafty	Chess program
197.parser	Natural language processing
256.bzip2	Data compression utility
168.wupwise	Quantum chromodynamics
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver in 3D potential field
178.galgel	Fluid dynamics: analysis of oscillatory instability
183.equake	Finite element simulation; earthquake modeling
189.lucas	Number theory: primality testing

Table 3: Benchmarks used in workloads.

each simulation, the workloads are randomly selected from the SPEC programs according to a uniform distribution.

3.4. Evaluation Metrics

In our experiments we assume that all the threads enter the system at the same time. As an evaluation metric, we use the speedup of the CMP configuration over the baseline performance of a single EV6 core. The speedup is computed in terms of global IPC, defined as the ratio between the global instruction count and the execution time.

3.5. Additional Assumptions

In this work we do not assume the presence of an oracle having a priori knowledge of the characteristics of the workload. Instead, the system must gather this information through thread migration and the cost of this operation is quantified and included in the evaluation. The only exception to this rule will be the best static assignment case, introduced for the purpose of comparison.

4. Benchmark Characterization

In this section we analyze the performance of the SPEC benchmarks when executing on EV5 and EV6 cores. The data are collected with the M5 simulator.

For each program, 2.5 billion instructions were executed on both EV5 and EV6 processor cores. The following statistics were collected: IPC, branch predictor accuracy, L1 data and instruction cache miss rate, and L2 miss rate. Moreover, execution was divided into windows of 1 million clock cycles and the values of the mentioned statistics within those windows of execution were also computed. This latter group of data, which we refer to as relative statistics, better captures the local variability of the programs behavior and will be used as input to the CMP simulator.

Figures 2 and 3 display the arithmetic mean and the variance of the relative IPC over program execution excluding a warm-up period of 500 million instructions. Three basic observations can be made. First, each program performs better on an EV6 than on an EV5 core. Second, the IPC

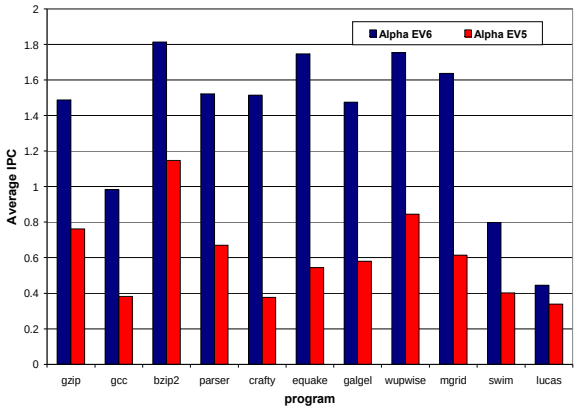

Figure 2: Average IPC.

exhibits a discrete variability across the benchmarks. Third, as the variance data point out, there are programs (e.g. *lucas*) which experience a large variation in IPC during execution and programs which exhibit more regular behavior (e.g. *crafty*, *equake*). Further analysis of the IPC traces over time highlights a correlation between IPC variation and program phases. As we will discuss in Section 6.3, these phases can be effectively exploited with dynamic thread assignment policy.

The analysis of branch predictor accuracy and cache miss rate led us to several conclusions. First, it confirmed how the considered collection of benchmarks exhibits heterogeneity in the exploitation of the different hardware resources. Second, it highlighted how the microarchitectural differences between the two cores contribute differently to the gain in IPC when moving from an EV5 to an EV6. Specifically, while the branch predictor accuracy does not have a significant increase in this transition, the miss rate strongly depends on the processor selection. However, the way the miss rate affects IPC varies significantly across the benchmarks. That is, cache miss rates cannot replace a reliable IPC as performance metric.

Figure 4 reports the ratios between the mean IPC value on an EV6 and on an EV5 processor. Note that, in the best case, an EV6 is 4 times as fast as an EV5. As mentioned, an EV6 core occupies 5 times the area of an EV5. Thus, a first analysis of this data suggests that, in case of static assignment, it is not efficient to use EV6 cores when the thread parallelism is high.

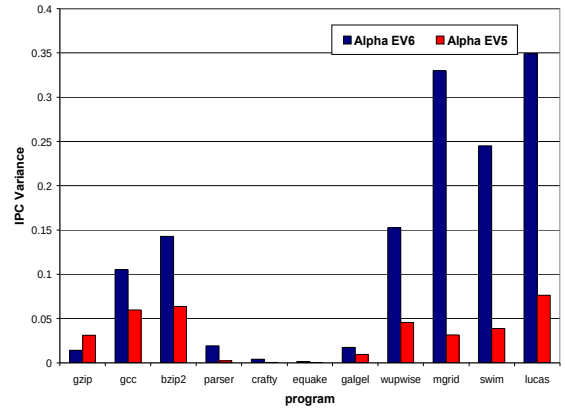

Figure 3: Variance of IPC.

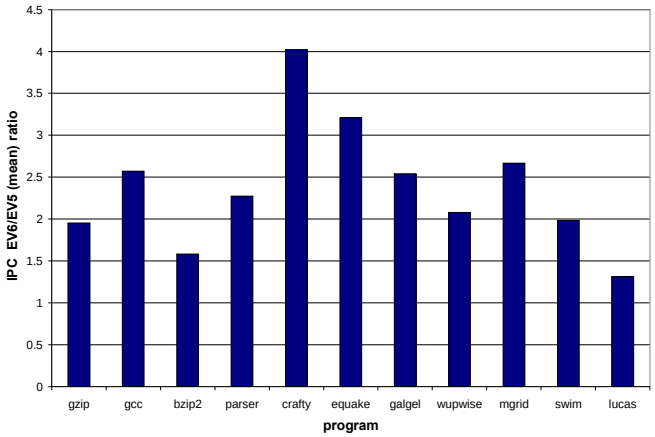

Figure 4: Ratio between average IPC on EV6 and EV5.

5. CMP Simulation Model

In this section we describe the CMP simulation model we use to evaluate different combinations of processors, workloads, and thread assignment policies.

5.1. The Model: Working Principles

The idea at the basis of the CMP simulator is the following. A multiprocessor system can be thought of as a collection of processor and thread objects where each thread represents an instance of one of the benchmark programs. At each simulation cycle, a thread can either be unassigned or associated to a core. Conversely, each processor can either be idle or running one of the available threads. System progress is simulated by having all the non-idle processors simultaneously advance a given number of clock cycles. Since the execution traces gathered with M5 provide the relative IPC, miss rate, and branch predictor accuracy computed each million clock cycle interval, each simulation tick on the CMP simulation model will also represent 1 million clock cycles of execution on the real system.

Each IPC value in an execution trace is used to determine the number of instructions that each thread will execute over a simulation period. Since IPC traces are collected for each (program, core) pair, the IPC data and the mapping information jointly drive the CMP simulations.

Finally, one more issue must be considered in order to have a correct model of the system. The input data to the CMP simulator are temporal sequences (a necessity given that all the cores are equally clocked). On the other hand, the current IPC of each program is clearly dependent upon the executing processor. As a consequence, for any given program, the values corresponding to a particular simulated clock cycle in the EV5 and in the EV6 IPC sequences relate to two different points of execution of the program itself. This is pointed out in Figure 5, which shows how the same instruction is temporally postponed on the (relatively slower) EV5 execution.

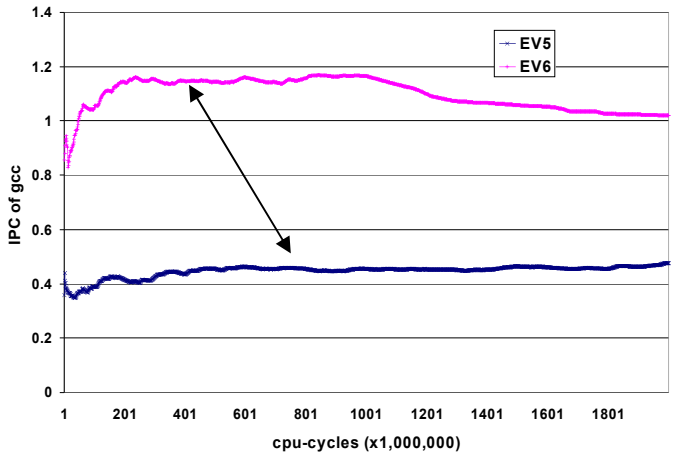

Figure 5: Location of the same instruction on the IPC temporal sequences corresponding to EV5 and EV6 cores.

To have the simulator work properly, each value of the traces must be associated with the program execution point it refers to. The execution point is expressed in terms of number of instructions previously committed. Upon thread migration, this information is used in order to locate the proper IPC value to use in the data sequence corresponding to the new core. Additional thread migration details are explained in the next subsection.

5.2. Modeling Thread Migration

A thread migration event can be thought of as an inter-core context switch; upon transfer, the architectural state (PC value, registers, etc.) of the migrating thread must be copied onto the destination processor. Copying the architectural state is required for correctness. For performance reasons, we must also consider the time needed to move and rebuild the state in other mechanisms such as caches.

One can imagine mechanisms to transfer the content of the L1 caches and the branch predictor state to the new core. Such mechanisms may be not trivial to implement. For example, the different size and associativity of the L1 caches may cause block evictions when moving from an EV6 to an EV5 core.

An easier way to think of the problem is to only move the architectural state of the migrating thread to the new processor. The branch predictor state and the cache content can be dynamically rebuilt during execution; hence, the new core will go through a warm up period of reduced throughput. We represent this by adding two parameters to the model: *switch_duration* and *switch_loss*. Upon thread migration, the IPC of the destination processor will be reduced by the *switch_loss* factor over a number of clock cycles indicated by *switch_duration*.

The *switch_duration* parameter is sized assuming that the cost for refilling the L1 caches is higher than the one for rebuilding the branch predictor state and transferring the thread architectural state. This is reasonable considering that the transfer of the architectural state can require few clock cycles, and that the branch misprediction penalty is 3 clock cycles compared to an L2 access time of 10 clock cycles. Assuming that the L1 caches must be completely refilled—the worst case assumption—it will take about 2048 and 514 accesses to the lower level cache to

refill the EV6 and the EV5 L1 caches, respectively. If most of the data can be retrieved in the L2 shared cache, 20480 clock cycles can be assumed to refill the bigger caches of the EV6. To be conservative and take into account the fact that L2 cache contentions due to thread migration may lengthen its access time, we'll assume a *switch_duration* of 1 million clock cycles.

6. Assignment Policies

In this section we discuss the working principles of different assignment policies, along with their advantages and limitations.

As mentioned in Section 2, a static assignment policy leads to a thread-to-core mapping that won't be subject to modifications over time. Conversely, a dynamic scheduling strategy exploits thread migration in order to better exploit available resources.

Two observations motivate the use of dynamic policies. First, as shown in Section 3 and as pointed out by previous work [7][8], the runtime behavior of any program tends to vary during execution. Thus, it may be beneficial to remap a thread to a different processor as the program phase changes. Second, thread migration can be seen as a mechanism to collect control information for performing the dynamic assignment itself.

In the remainder of the section we describe the different scheduling mechanisms in detail.

6.1. Static Assignment

Finding the best static assignment of jobs to processors is a well studied problem both for homogeneous and heterogeneous architectures [10][11]. Known methods are based on the assumption that the characteristics of the workload are known a priori. Moreover, since the scheduling problem is NP-hard, the proposed solutions rely on heuristics which seek sub-optimal solutions. In our simulation model, two static scheduling mechanisms are implemented: a random and a pseudo best static assignment.

The random static assignment is based on the assumption that the system has no a priori knowledge of the workload characteristics and therefore assigns threads to processors in a random fashion. The assignment tries however to maximize the use of the EV6 cores, which will be the first ones to be assigned a thread. In other words, the policy won't let EV6s idle unless their number exceeds the number of threads to be assigned. If the system has more programs to be scheduled than available cores, unassigned threads will be given a processor as soon as one is available. In our simulated evaluations, the effect of particularly unlucky random scheduling is mediated by the fact that each data point is on average produced by running 100 distinct simulations over random thread selections and assignments.

The pseudo best static assignment assumes that the runtime characteristics of the threads to be executed are known a priori. In particular, it assumes the presence of an oracle providing the system with the program's IPC on both cores and the number of instructions to be executed. Note this policy is only relevant for purposes of comparison.

Implementing the best static assignment would mean exploring all the possible permutations of assignments of threads to cores. Instead of doing this, we use a simple heuristic to find a suboptimal assignment. In the simulation phase, the suboptimality will also be mediated by averaging the results on randomly selected workload configurations.

The adopted heuristic is based on two observations. First, the global IPC is negatively affected by a long execution of a slow thread on an EV5 processor. Second, an optimal

assignment tends to exploit the EV6 cores by having them execute more threads. Specifically, as the EV6 cores are on average twice as fast as the EV5 ones, the heuristic will attempt to schedule twice as many threads on an EV6 as on an EV5. In order to better follow these two principles, the assignment policy will additionally sort the threads basing on their IPC on the two processor types and try to assign to a core the first available thread which exhibits the highest IPC on it.

6.2. Round Robin Dynamic Assignment

Besides not being able to capture the phase behavior of the executing programs, static assignment has two main drawbacks. First, it does not optimize EV6 usage. In fact, when one core becomes idle, it will persist in that state unless some unassigned threads exist. Secondly, the execution of "slow" threads on EV5 cores may penalize overall system performance, especially if global IPC is taken as a performance metric.

A simple round robin dynamic assignment can be used to limit the two effects above. By periodically rotating the assignment of threads to processors in a round robin fashion, this policy ensures that the available EV6 cores are equally shared among the running programs. Moreover, when all the threads are assigned and some EV6 cores become idle, jobs are moved from the running EV5 to the free EV6 cores, diminishing their inactivity.

In this scheme, a *swap_period* parameter defines the frequency of the rotation. At the end of each rotation period as many threads as possible are migrated in parallel. Note that, since there are fewer EV6 than EV5 cores, several swap cycles will be necessary to have a complete rotation of the threads.

The round robin strategy is blind: it works unaware of the runtime behavior of the threads and does not use run-time information in order to drive the reassignment. Nevertheless, it represents an improvement in respect to a simple static assignment.

6.3. IPC-Driven Dynamic Assignment

We can improve on this dynamic assignment policy by considering the characteristics of the executing threads. In particular, the assignment policy should try to ensure a good assignment of threads to cores over the whole execution, causing dynamic changes as the programs enter different execution phases.

As mentioned in Section 3, relative IPC is a good metric to quantify thread behavior and drive the dynamic assignment. In fact, the goal of the assignment is to maximize the overall IPC, which in turns depends on the IPC of the individual threads. Moreover, the value of the IPC of the running threads can be easily made available at each execution cycle.

The idea at the basis of the IPC-driven assignment can be stated as follows. The overall performance of a heterogeneous system can be optimized if, at any given instant of execution, the threads which benefit more from the architecture of the most sophisticated processors are executed on them. Conversely, the threads that achieve only a modest performance increase by executing on the "faster" cores can be run on the "slower" ones and be migrated just when a faster processor becomes idle. Hence, for each thread, the ratio between the IPCs on an EV6 and on an EV5 processor can be used to guide the assignment: threads with higher ratios will run on EV6s and, conversely, threads with lower ratios will be executed on EV5s. This approach has the benefit of simplicity: it does not require the evaluations of all the possible permutations of assignments of threads to cores but it implies a sorting operation.

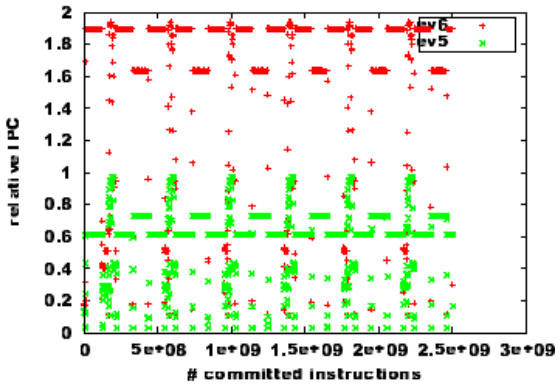

Figure 6: *mgrid* relative IPC.

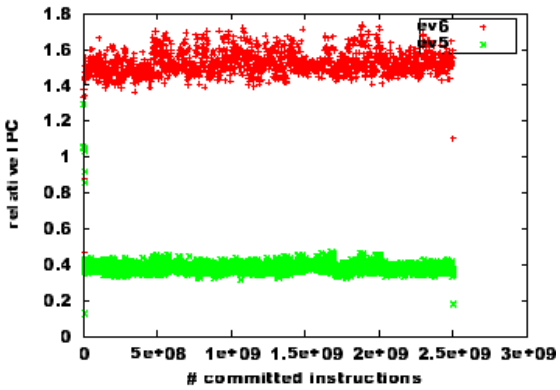

Figure 7: *crafty* relative IPC.

Two important issues must be addressed in order to implement this policy. First, the IPC values on both processors must be available in order to make assignment decisions. Since we don't assume the presence of an oracle providing such information, some learning mechanism must be established. This mechanism must take into account variations of the IPC over program

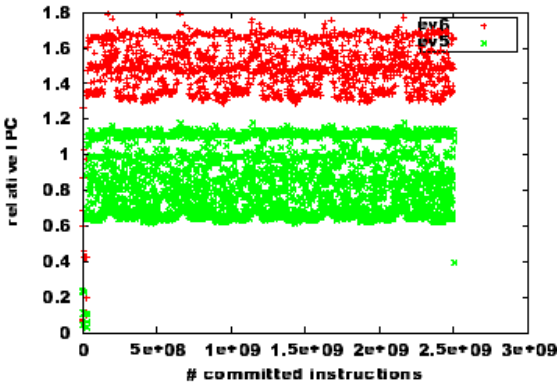

Figure 8: *gzip* relative IPC.

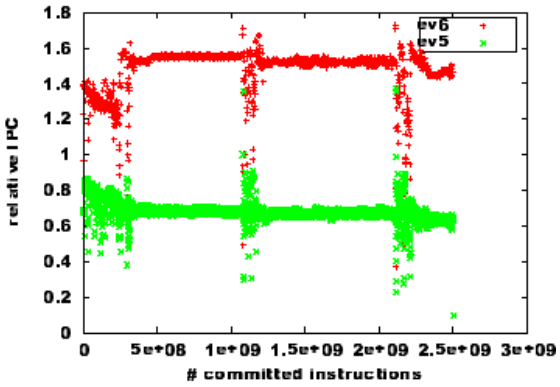

Figure 9: *parser* relative IPC.

execution. Second, it must be defined how and how often the control information must be used.

It can be assumed that a program's current IPC is always available for the processor that is executing the program; since the program only executes on one processor at a time, however, the IPC for the other processor is unknown. Thus, for each running thread, a migration action is needed in order to refresh the aforementioned IPC ratio. The system will experience forced thread migrations when an update of the control information is necessary and IPC-driven migrations when the thread-to-core assignment must be rebalanced.

6.3.1. Controlling Thread Migrations in IPC-Driven Assignment

Forced migrations performed in order to keep the IPC estimate accurate can be initiated in two ways: periodically on all threads at the same time, or on a per-thread basis. Figures 6-9 show the variation of the relative IPC during execution for several benchmarks. As can be seen, each program has a unique IPC phase behavior. The patterns differ in both shape and phase duration. Thus, no one migration period will suit all programs. For example, long periods would penalize threads with shorter execution phases (e.g. *mgrid*), while short periods would trigger useless migrations on more stable execution patterns (e.g. *crafty*).

Previous work shows that the phase behavior of the programs does not depend upon the executing processor [8]. This same fact can be observed in Figures 6-9: for any benchmark, sudden variations in IPC can be observed at the same execution points on both processors. According to this observation, forced thread migrations can be triggered by a rapid variation in IPC. Tailoring the controlled swap operations to the single threads, this approach minimizes the number of performed thread migrations.

The use of the bare relative IPC as control variable would cause continuous variations of IPC within a limited range to trigger frequent migrations. As it appears from Figure 8, this would be the case in *gzip*. In order to avoid this phenomenon, the IPC-driven dynamic policy uses the moving average of the IPC as the control variable. Specifically, the moving average is computed summing its previous value weighted by a factor 0.35 to the current value of the IPC weighted by a factor 0.65. Additionally, a forced migration of all the threads is triggered at the beginning of execution (after a warm-up period) in order to initialize the system.

IPC-driven migrations will be executed by constantly comparing the biggest IPC ratio on the EV5 processors with the smallest on the EV6 processors. If the former exceeds the latter, a swap of the corresponding threads will be triggered. A *swap_inactivity* period parameter is introduced in order to limit the compare operations and allow the system to stabilize between two

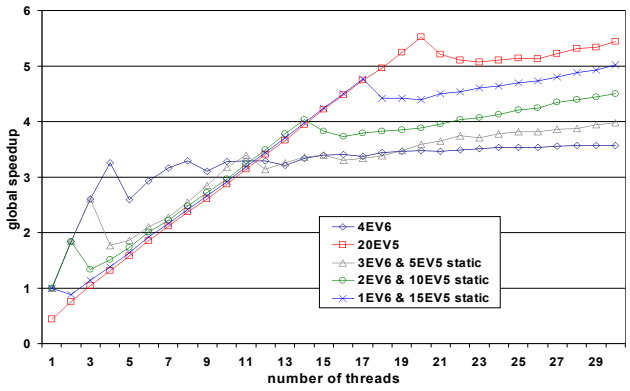

Figure 10: Comparison between homogeneous and heterogeneous system with static random assignment.

consecutive swap operations. In particular, this parameter defines a minimum number of clock cycles between two consecutive migrations of the same thread.

Finally, as in the round robin case, this dynamic policy will prevent EV6 cores from being idle by migrating unassigned or EV5 threads to them.

7. Simulation Results

In this section we present the results of our simulations. We first evaluate the static assignment policies on homogeneous and the heterogeneous systems, and then show how the use of dynamic policies outperforms any static configuration. As previously mentioned, each experimental data point is obtained by averaging the results of 100 simulations run over random thread selections and initial assignments.

7.1. Homogeneous vs. Heterogeneous Configuration with Static Assignment

Figure 10 compares the homogeneous and the heterogeneous configurations when a random static assignment is used. The intersection between the two curves corresponding to the homogeneous configurations determines two areas: if the workload presents low thread-level parallelism (less than eleven threads) then a 4EV6 configuration is preferable, otherwise a 20EV5 setup is more effective.

All the curves present a drop when the number of threads exceeds the number of available processors; at this point queuing develops in the system. This effect is mitigated by the introduction of additional threads, which allows a better balancing of the cores utilization thus diminishing their idle time.

The analysis of Figure 10 shows that if mechanisms to optimize the thread-to-core assignment are not used, the utilization of a heterogeneous setup does not bring any benefit. In fact, the global IPC can be negatively affected by the execution of slow threads on EV5 cores. Moreover, the EV6 utilization is not maximized.

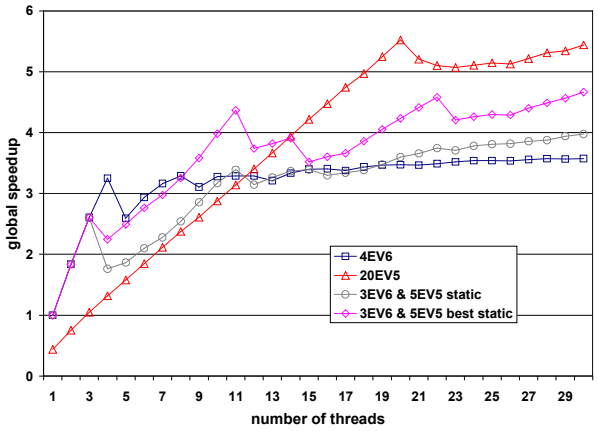

Figure 11: Comparison between random and pseudo optimal static assignment.

This fact becomes evident by the observation of Figure 11 which compares the random and the pseudo optimal static assignment for a 3EV6-5EV5 configuration. Not only does the best static assignment represent an average of 20% improvement over the random one, but it is also more effective than a 4EV6 configuration for nearly all scenarios with a non-negligible degree of thread parallelism. A similar improvement is achieved with all three heterogeneous configurations tested (the curves have been omitted for readability).

7.2. Dynamic Assignment

Figure 12 compares the performance of the Round Robin assignment policy to the configurations exhibiting the best performance in the previous experiments.

The results permit several observations. First of all, on a 3EV6-5EV5 configuration the Round Robin strategy performs similarly and generally better than the pseudo optimal static assignment policy. It should be noted that the latter assumes an a priori knowledge of the IPC of the running threads, while the former is completely unaware of the characteristics of the workload. Hence, a great benefit is gained by simply reassigning the idle EV6 cores and balancing the EV6 utilization among the running threads. Second, notice that all the dynamic strategies either perform better than, or approximate (1EV6-15EV5), the best static assignment policy on a 3EV6-5EV5 configuration even when the number of threads is limited. This consideration and a comparison with Figure 11 leads to the conclusion that dynamic assignment is preferable or comparable to the ideal static one, independent of the heterogeneous configuration even for low degrees of thread parallelism. Finally, in case of a high degree of thread-level parallelism, the dynamic assignment on heterogeneous configuration can be beaten only by a 20EV5 configuration. However, the presence of a single EV6 core still guarantees better performance than the homogenous case for fewer than 14 threads, and the 2EV6-10EV5 configuration still allows comparable performance up to 30 threads. Thus, the heterogeneous solution appears to be attractive even for high degrees of thread parallelism.

In Figure 13 we compare the Round Robin and the IPC-driven dynamic policies, and include the homogeneous configurations for comparison. The rotation period used in case of the Round Robin strategy is 500M clock cycles. However, experiments where this period was varied from 50M to 800M clock cycles did not show dramatic differences in the results.

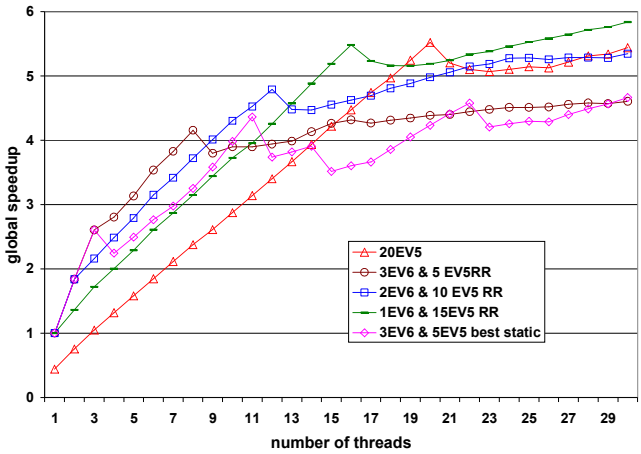

Figure 12: Comparison between Round Robin dynamic policies and static assignment.

We can observe that the IPC-driven strategy brings a little performance improvement over the Round Robin one, which becomes more manifest when the number of simulated threads exceeds the number of available processors. Thus, the use of the runtime characteristics of the system is beneficial even at the cost of maintaining the control information.

However, the incremental performance improvement of the IPC-driven policy over the Round Robin one highlights that, for the simulated workloads, a simple dynamic scheduler periodically triggering thread migration and unaware of the runtime behavior of the running threads is sufficient for exploiting the benefits of a heterogeneous system. Moreover, this holds particularly with a limited degree of thread-level parallelism. In order to understand this fact, some deeper analysis is needed.

The use of a dynamic policy contributes a better utilization of the system in two ways: (i) by keeping the higher performance processors fully utilized, (ii) and by better distributing the load across cores of different types. Note that the latter is done differently depending on the dynamic

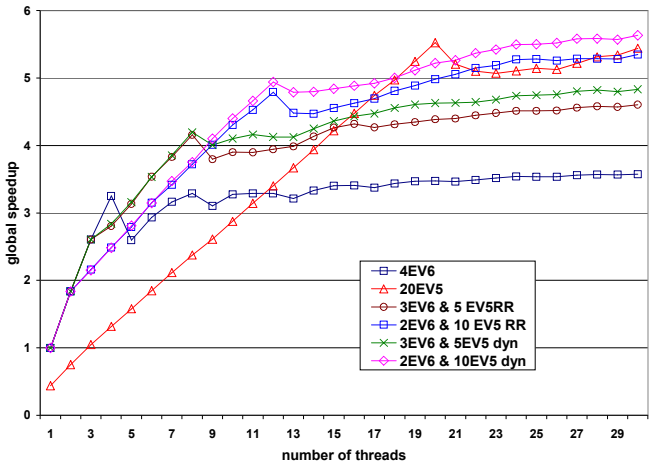

Figure 13: Comparison between dynamic policies and homogeneous configurations.

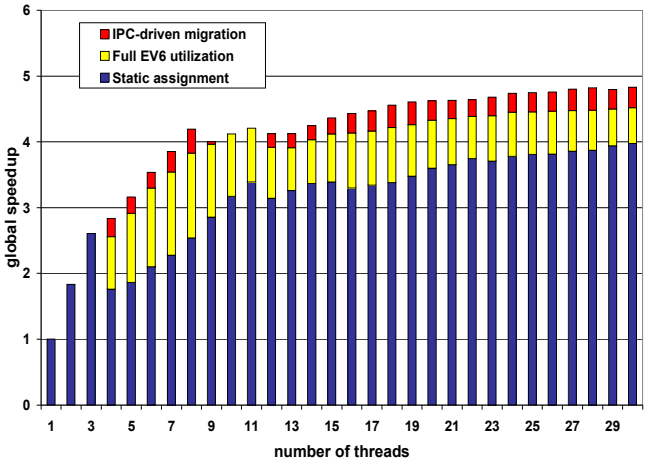

Figure 14: Components of speedup: IPC-driven assignment.

policy used. Specifically, the Round-Robin policy ensures a fair share of EV6 utilization to each thread through the periodic migrations; conversely, the IPC-driven allocates EV6 cores to threads which can most benefit from them.

Figures 14 and 15 show how these two factors affect the speedup of each dynamic policy over the static assignment. The displayed graphs correspond to the 3EV6-5EV5 configuration; similar results have been obtained in case of the other considered heterogeneous setups.

It can be noted that the full utilization of EV6 cores accounts for the higher contribution to the overall speedup, independent of the policy and the level of multithreading considered. Moreover, we can divide the graphs into 4 regions based on the number of active threads. If the number of threads is less than the number of available EV6s (that is, 3), then no migration and no performance improvement over the static case takes place. If the number of threads does not exceed the number of available cores, then the EV6 full utilization accounts for about 30% of the speedup and the load distribution for between 7% and 10%. This happens independently of the

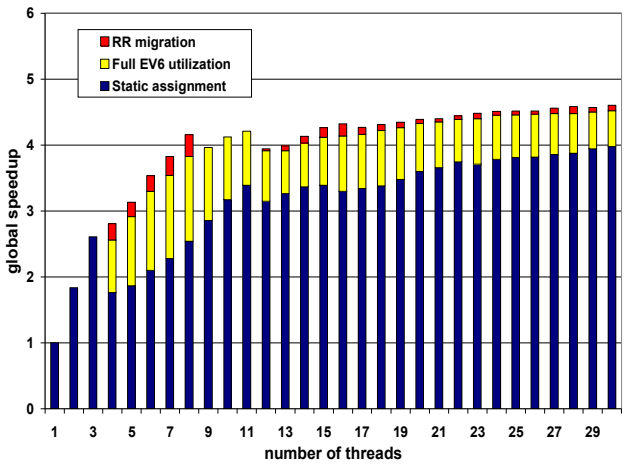

Figure 15: Components of speedup: Round Robin assignment.

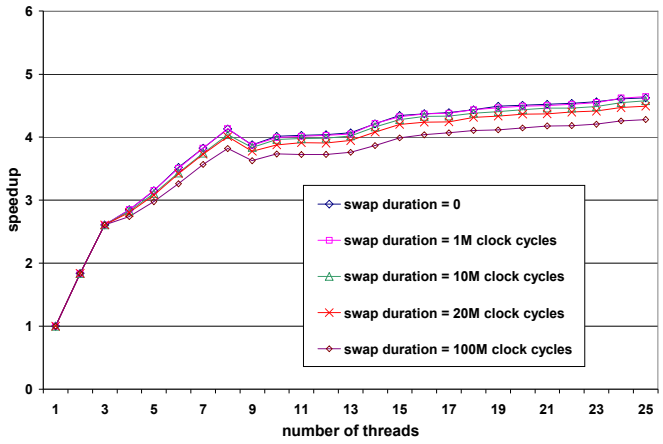

Figure 16: Effect of cost of thread migration on overall performance.

mechanism used to provide dynamicity. If the number of threads is slightly greater than the number of processors (namely, between 9 and 11), then a 20% speedup is provided exclusively by fully utilizing the EV6 cores. Finally, for higher degrees of thread parallelism, EV6 full utilization accounts for about 15% of the speedup; IPC-driven migration brings an additional speedup of 7% while the increment given by periodic thread migration in case of Round Robin policy is around 3%.

Figure 16 shows how the cost of thread migration affects the performance with dynamic, IPC-driven assignment. In particular, the result of varying the swap duration parameter is displayed. We observe that as we vary the cost of thread migration from 0 to 100 million cycles, there is no significant reduction in speedup. This is a consequence of making migration decisions on a per-thread basis; tailoring thread migration to the behavior of single programs (instead than forcing it in a system wide manner) minimizes the number of migrations experienced by the system.

As a general conclusion, we can observe that when a dynamic assignment policy is used, any heterogeneous configuration can accommodate a variety of degrees of thread parallelism better than a homogeneous one. The improvement in extreme cases (few threads and 20EV5 or many threads and 4EV6) can reach 60-80%. Moreover, the modest performance improvement of the IPC-Driven compared the Round Robin assignment policy is due to the fact that the mechanism which the most account for the speedup over the static assignment is shared by the two policies. This consideration suggests that, especially in case of low level of thread parallelism, a simple dynamic mechanism is sufficient for exploiting the benefit of heterogeneous CMP. In fact, the difference in IPC ratio across the considered workloads is sufficient to benefit from dynamic assignment but not enough to motivate sophisticated migration policies.

8. Analytical Model

In this section we present an analytical model of a CMP system. The goal is to generalize the discussion to different processor types and programs, and to be able to answer the following question: How different should the processor cores and benchmark programs be in order for a heterogeneous configuration to be preferable to a homogeneous one?

Parameter	**Description**
Processor type = {SP, FP}	SP=slow processor FP= fast processor
CMP configuration = {HOM-SP, HOM-FP, HET(*n*)}	HOM-SP=homogeneous with SP HOM-FP=homogeneous with FP HET(*n*) = heterogeneous with n FP
a_{SP}	area of SP
a_{FP}	area of FP
$\alpha = a_{FP}/a_{SP}$	area factor
$IPC_{SP}(p)$	IPC of program p on SP
$IPC_{FP}(p)$	IPC of program p on FP
$\beta(p) = IPC_{FP}(p)/ IPC_{SP}(p)$	performance factor for program p
k	number of SP on HOM-SP
$\lfloor k/\alpha \rfloor$	number of FP on HOM-FP
n	number of FP on HET(*n*)
$\lfloor k - n\alpha \rfloor$	number of SP on HET(*n*)

Table 4: Parameters in the analytical model.

The parameters of our model are listed in Table 4. Specifically, the model assumes two types of processor cores: *slow* (*SP*) and *fast* (*FP*). These cores can be arranged in a homogeneous (*HOM-SP* and *HOM-FP*) or a heterogeneous (*HET*) configuration. To allow a fair comparison between the different CMP configurations, our chosen configurations all occupy the same amount of on-chip area. To this end, we introduce two area parameters a_{SP} and a_{FP} and an area factor α. If *k* represents the number of *SP* in a homogeneous configuration and *n* the number of *FP* in a heterogeneous one, then α is used to determine the number of *FP* in a *HOM-FP* and the number of *SP* in a *HET(n)* in order to keep the die area constant.

The two processor cores may differ in various ways. They may be architecturally different, and use a distinct cache configuration, as is the case of the Alpha EV5 and EV6 cores considered above. Or, they may share the same architecture but be clocked at different rates. Clearly, in the latter case the memory bandwidth available to the fast processors must also be scaled accordingly. In order to encompass different scenarios, our model represents the diversity between the processors in terms of their performance. Specifically, $IPC_{SP}(p)$ and $IPC_{FP}(p)$ are the average IPC reported by a program *p* on a uni-processor configuration consisting of a single slow and a single fast processor, and $\beta(p)$ is ratio between the two, that is, the relative performance improvement on the fast core.

Notice that we could have represented the performance also as a function of the time, *t*, in the form *IPC(p,t)*. In our simplified model, we avoid the use of the time dimension, as this analysis has already been covered in the simulation model presented in the first part of the paper. This choice implies the assumption that the influence of the program phase behavior on the overall performance of a heterogeneous system is a small factor compared to the ability of a

heterogeneous system to equally distribute the fast processor utilization. This agrees with the detailed simulation results reported in Section 7.2 (Figure 14 and 15).

The performance equations for the three CMP configurations are as follows.

$$T_{HOM\_SP}(\tau) = \begin{cases} \max_t \left(\frac{I}{IPC_{SP}(t)} \right), \tau \leq \\ \left\lceil \frac{\tau}{k} \right\rceil \frac{I}{E[IPC_{SP}(t)]}, \tau > \end{cases} \qquad T_{HOM\_FP}(\tau) = \begin{cases} \max_t \left(\frac{I}{IPC_{FP}(t)} \right), \tau \leq \left\lfloor \frac{k}{\alpha} \right\rfloor \\ \left\lceil \frac{\tau}{\left\lfloor k/\alpha \right\rfloor} \right\rceil \frac{I}{E[IPC_{FP}(t)]}, \tau > \left\lfloor \frac{k}{\alpha} \right\rfloor \end{cases}$$

$$T_{HET}(n,\tau) = \begin{cases} \max_t \left(\frac{I}{IPC_{FP}(t)} \right), \tau \leq n \\ \max_t \left(\frac{I\tau}{nIPC_{FP}(t) + (\tau - n)IPC_{SP}(t)} \right), n < \tau \leq n + \lfloor k - n\alpha \rfloor \\ \left\lceil \frac{\tau}{n + \lfloor k - n\alpha \rfloor} \right\rceil \frac{I(n + \lfloor k - n\alpha \rfloor)}{E[nIPC_{FP}(t) + \lfloor k - n\alpha \rfloor IPC_{SP}(t)]}, \tau > n + \lfloor k - n\alpha \rfloor \end{cases}$$

In particular, $T_{CMP\_CONF}(\tau)$ indicates the execution time of τ threads, each consisting of I instructions, on configuration *CMP_CONF*. We assume that the τ threads are randomly drawn from the set of available benchmarks according to a uniform distribution.

The behavior of the two homogeneous configurations is similar. Specifically, if the number of threads does not exceed the available processors, then the execution time is determined by the slowest thread. Otherwise, queuing originates in the system and the average length of the queue is given by the ratio between the number of threads and the number of processors. The average execution time is the product between the average execution time of a thread and the length of the queue.

In the case of a heterogeneous configuration, since our analytical model neglects program phases, a round-robin assignment policy is assumed (this is a fair assumption given the marginal improvements reported in Section 7.2). Three possible scenarios can occur. If the number of threads is lower than that of the fast processors, then no reassignment takes place and the execution time corresponds to that of a *HOM-FP* configuration. Otherwise, a periodic rotation takes place and each thread is assigned a fair share of the available fast processors. The average IPC of a thread is given by the weighted average between the IPC_{FP} and IPC_{SP} of the program itself, the weights being the number n of *available FP* and the number of *used SP* (which, in turn, depends on the number of running threads). If the number of threads exceeds that of the processors, all the *SP* are utilized, queuing originates in the system, and the execution time is given by the product between the average size of the queue and the average execution time of a single thread.

For the sake of generality, the model can be refined through the introduction of a performance degradation factor δ accounting for increased memory and interconnect contention. Ideally, δ should be a function of the number of processors ($n + \lfloor k - n\alpha \rfloor$). Since this factor is negligible in the scenarios simulated in this paper, it has been omitted in the performance equations.

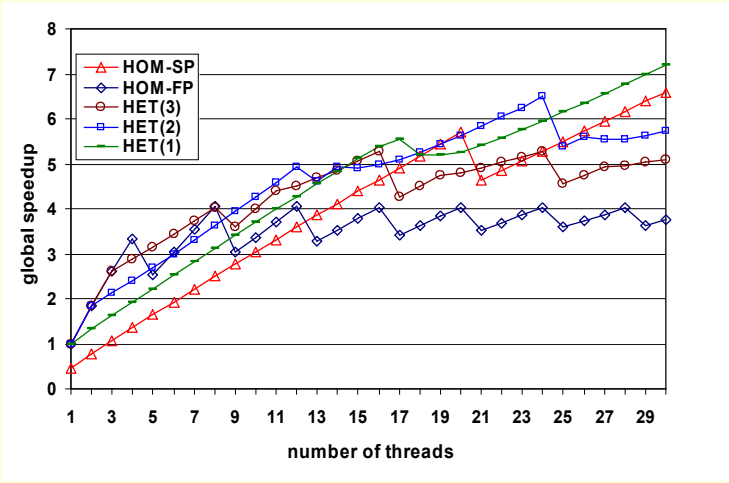

Figure 17: Speed-up obtained by applying the analytical model, k=20, to the benchmark in Table 3.

8.1. Model Validation

In order to validate our analytical model, we tested it on the benchmark programs in Table 3. Specifically, we assumed the Alpha EV5 to be the *SP* and the Alpha EV6 to be the *FP*. As $IPC_{SP}(p)$ and $IPC_{FP}(p)$, we considered the *average* IPC reported by program p during the simulation of 2.5 billion instruction on the corresponding processor core. Note that the area factor α is equal to 5. According to the simulations performed in the first part of the paper, we set the parameter k to 20 and, for n, we tested the values 3, 2 and 1. Finally, the number of threads τ varies from 1 to 30.

The results produced by the analytical model are shown in Figure 17. As done previously, we show the speedup as compared to a uni-processor configuration consisting of a single *FP*. Note that the aspect and the trend of the curves are very similar to what produced in our simulations (Figures 11 and 12). The bigger discontinuity (i.e., the zig-zag behavior of the graphs) is due to the fact that we are considering average IPC values and not parameters which are functions of time.

8.2. Additional Experiments

In this section, we use our analytical model to derive general conclusions for different processor types. In particular, we study the relationship between speedup and the degree of difference between the processor types. We focus on two cases: *i*) *SP* and *FP* are architecturally the same but are clocked at different rates, and ii) *SP* and *FP* are architecturally different.

8.2.1. Different Clock Rates

We first accelerate the *FP* clock frequency by a constant factor β, that is, $f_{FP}= \beta f_{SP}$, f_{FP} and f_{SP} being the two clock frequencies utilized. If we report the IPC in reference to the slower clock, we can say $IPC_{FP}(p)=\beta * IPC_{SP}(p)$ for each program p. This implies that the fast processor is provided with enough memory bandwidth to sustain the increased fetch rate. Moreover, since the two processors are architecturally the same, we assume that the area factor α is 1.

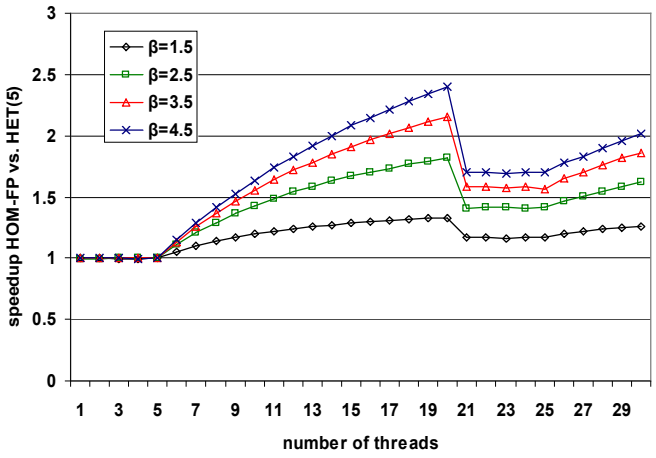

Figure 18: Comparison between *HOM-FP* and *HET(5)* with k=20 and α=1.

We performed evaluations by varying β and n, that is, the number of processors the increased clock cycle is applied to. Note again that the required memory bandwidth of the chip increases with both factors. In Figures 18 and 19 we show the results obtained by setting n to 5 and 10, respectively, and varying β between 1.5 and 4.5. For each value of β, we show the speedup obtained by accelerating all the processors versus only a subset (n) of them. We make the following observations.

Obviously, if the number of threads does not exceed that of *FP*, the two configurations perform the same. The maximum difference is observed when the number of threads is greater than that of *FP* but less or equal than that of processor cores. However, we can observe that the maximum speed-up of *HOM-FP* versus *HET(5)* is between 1.3 and 2.4, and the one of *HOM-FP* versus *HET(10)* is between 1.2 and 1.65. Moreover, when there are more threads than processors,

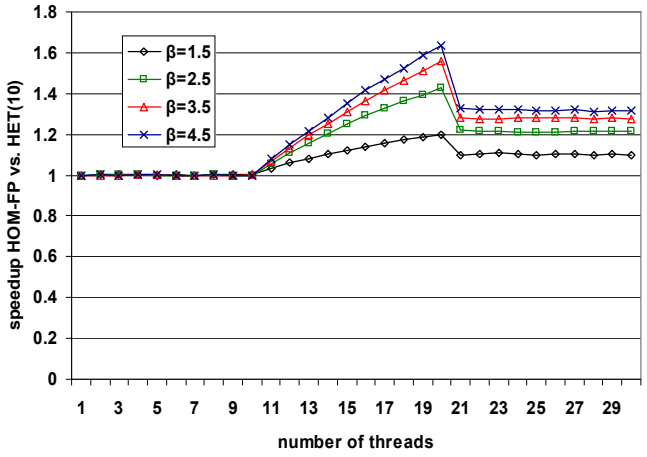

Figure 19: Comparison between *HOM-FP* and *HET(10)* with k=20 and α=1.

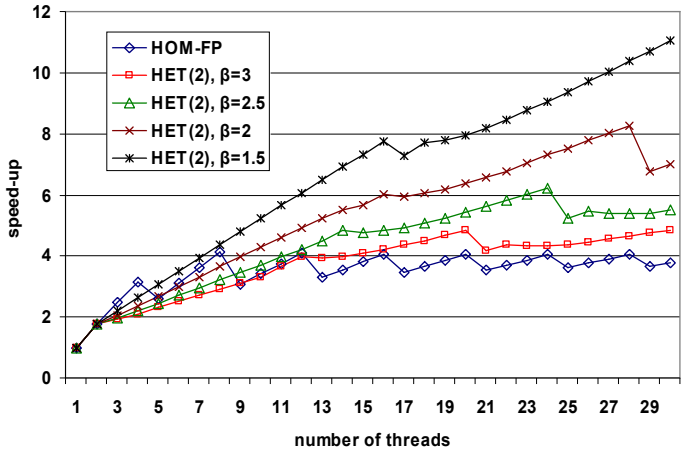

Figure 20: Speedup against a single FP with β/α=0.5.

the difference between the two configurations is less significant.

As a conclusion, applying a faster clock to only a subset of the cores in a CMP reduces power consumption without implying a significant performance loss in throughput-oriented workloads, especially if the number of threads does not approximate the number of processors. We note also that the relative advantage of faster clocks decreases with increased clock frequency. That is, the effectiveness of accelerating all processors, rather than a subset of them, diminishes as the accelerated clock frequency increases. As an example, for n equal to 5, the speedup/β ratio of 2.4/4.5 is less than 1.3/1.5.

8.2.2. Different Architectures

We now consider the scenario where the two processors are architecturally different. In this context, we assume that a higher performance factor is achieved at the cost of a higher architectural complexity, which translates into a higher area factor, α. This is the case of the Alpha EV5 and AlphaEV6 cores considered above.

To this end, we performed different experiments by varying β and, at the same time, keeping the ratio β/α constant. In particular, in Figure 20, β/α is set to 0.5 and β varies from 1.5 to 3. A homogeneous configuration is compared to a heterogeneous configuration with two fast processors. The number of slow processors is dependent upon α, which, in turns, grows with β. The speedup as compared to a uni-processor configuration consisting of one *FP* is reported.

It can be observed that, in general, keeping the value of β lower leads to a higher overall speedup, especially for high multithreading degrees. This has two explanations. First, as β increases, α also increases. Therefore, in the same area it is possible to accommodate fewer processors. If the number of threads is high, the performance increase on a few *FP* does not make up for the smaller number of available cores. Second, the speedup is reported as compared to a *FP* with the same β as the one of the simulated CMP configuration. Thus, the execution time of the baseline configuration decreases with β, too.

Similar trends have been reported with different values of β/α (namely: 0.75, 1). As a result, when considering architecturally different processors it is essential to relate the performance improvement of the *FP* to its cost, in this case reported in terms of area.

9. Related Work

Prior related work can be classified in two categories: work explicitly referring to heterogeneous CMP architectures, and work addressing the thread-to-core scheduling problem in a theoretical way.

In [12] the authors address the impact of heterogeneity on distributed shared memory systems, and assume the presence of few nodes with large caches for supporting single-thread parallelism, and many nodes with smaller caches for multi-thread parallelism. Static assignment policies to map jobs to processors are assumed.

In [3] resource sharing between adjacent cores on a CMP is studied as a mean for saving area die and increasing the overall system performance. In [1] it is studied how to dynamically map threads to cores in a CMP in order to reduce power consumption.

In [9] a method to automatically synthesize a custom architecture and in parallel assign jobs to cores is proposed. The heterogeneity is achieved by augmenting the instruction set of homogeneous processors via custom instructions.

The work presented in [11] proposes a static heuristic for heterogeneous processors based on the use of an acyclic precedence graph. However, the use of this heuristic requires an a priori knowledge of the characteristics of the workload.

The two pieces of work closest to the one presented in this paper are [2] and [17]. We borrow from [2] the choice of the processors and the homogeneous configurations. However, there are several crucial differences, both conceptual and methodological. First, the authors of [2] use system-wide profile-run phases in order to periodically compute the best thread-to-core assignment and apply it. In each profile phase, common for all the programs, the collected statistics are used in order to compute the assignment to be used during the next run phase. This is done by evaluating a subset of the possible permutations of assignments of threads to processors. Instead of this system-wide arrangement, our approach is thread specific, and thread reassignment is driven by changes in IPC during execution. Avoiding the evaluation of permutations of assignments allows us to reduce the number of forced thread migrations. Second, the authors of [2] do not explicitly address or quantify thread migration. In fact, their approach to dynamicity can be thought as performing a sequence of periodic static assignments according to metrics collected during the profile phases. Third, in [2] the Round Robin policy is not considered. Since, at the end of each profile phase, the policies proposed distribute the load in order to maximize the throughput, they behave similarly in principle to the IPC-driven assignment. We crosschecked this by implementing them in our simulator and observed no substantial performance differences. Fourth, in [2] the evaluation is restricted to the low thread parallelism working region. Note that it is specifically in this region that the simple Round Robin policy performs like the more sophisticated IPC-based one. Finally, our work provides an analytical evaluation identifying the factors of the dynamic policies that provide the speedup over the static approach.

In [17] an analytical model motivating the use of heterogeneous multiprocessor systems is presented. The authors assume the presence of a slow and a fast processor. However, there are several differences between their approach and ours. The most substantial concerns the workload, which ultimately affects the motivation for introducing heterogeneity. In fact, while we consider a heterogeneous workload consisting of different programs with varying characteristics, the authors of [17] imagine deploying a single program on the analyzed system. Specifically, they assume that the program is partially parallelizable, and distinguish its serial fraction f_S from its parallelizable fraction f_P. Having many slow processors allows the parallel execution of f_P,

whereas the fast processor provides a speedup to f_S. Clearly, a mechanism to separate a program into its parallel and its serial portions and to deploy them accordingly is needed. This assumption is not required in our analysis. A second difference relies on the fact that, while we basically present a performance/area model (in that we compare CMP configurations occupying the same on-chip area), in [17] a performance/cost analysis is presented. The cost, however, is not quantified in practical terms. It is worth noticing that the study in [17] refers to generic multi-computer environments, whereas we specifically relate to chip multiprocessor systems, where a performance/area analysis is more meaningful.

10. Conclusions

Heterogeneous multiprocessor systems have been recently proposed as a mean to efficiently accommodate different degrees of thread parallelism and to meet the needs of multi-programmed computing environments. In fact, the presence of many low area cores ensures a high level of parallelism and the one of few high performance cores guarantees high throughput when thread parallelism is low. Furthermore, threads with distinct hardware resource requirements can be effectively mapped to processors with different complexity.

In this work we argue that the benefits of heterogeneous CMP are increased by the use of dynamic policies for assigning threads to processors. Not only do such strategies better capture the dynamic behavior of the running threads, they also maximize the usage of the high performance cores.

In this work we propose two different dynamic policies and evaluate them on three distinct heterogeneous configurations of cores having the same ISA. We compare their performance with the one provided by a random and a pseudo optimal static assignment policy and by two homogeneous configurations. The evaluation is performed for distinct degrees of thread parallelism. Our analysis is based on the use of an own simulation model representing the cost of thread migration and of all the required control mechanisms.

Our results highlight the ability of all analyzed heterogeneous configurations to offer good performance across several degrees of thread parallelism when a dynamic policy is used. A dynamic policy on a heterogeneous CMP can outperform a random assignment policy by 20% to 40% and a homogeneous configuration by 20% to 80% depending on the number of threads simulated.

Acknowledgements

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

References

[1] R. Kumar, *et al*. “Single-ISA Heterogeneous Multi-core Architecture: The Potential for Processor Power Reduction.” *Proc. 36th Int’l Symposium on Microarchitecture*, pp. 81-92, 2003.

[2] R. Kumar, *et al.* "Single-ISA Heterogeneous Multi-Core Architecture for Multithreaded Workload Performance." *Proc. 31st Int'l Symposium on Computer Architecture*, pp. 64-75, 2004.

[3] R. Kumar, N.P. Jouppi, and D.M. Tullsen. "Conjoined-core Chip Multiprocessing." *Proc. 37th Int'l Symposium on Microarchitecture*, pp. 195-206, 2004.

[4] J. Hennessey and D. Patterson. *Computer Architecture a Quantitative Approach*. 3rd Edition, Morgan Kauffmann Publishers, Inc., 2003.

[5] R.E. Kessler, E.J. MecLellan, and D.A. Webb. "The Alpha 21264 Microprocessor Architecture." *IEEE Micro*, vol. 19, no. 2, pp. 24-36, March/April 1999.

[6] N.L. Binkert, E.G. Hallnor, and S.K. Reinhardt. "Network-Oriented Full-System Simulation using M5." *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, (CAECW 03), February 2003.

[7] T. Sherwood, *et al*. "Discovering and exploiting program phases." *IEEE Micro*, vol. 23, no. 6, pp. 84-93, 2003.

[8] T. Sherwood, S. Sair, and B. Calder. "Phase Tracking and Prediction." *Proc. 30th Int'l Symposium on Computer Architecture*, pp. 336-349, 2003.

[9] F. Sun, *et al*. "Synthesis of Application-specific Heterogeneous Multiprocessor Architectures using Extensible Processors." *Proc. 18th Int'l Conf. on VLSI Design*, pp. 551-556, 2005.

[10] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architecture." *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp.175-187, Feb. 1993.

[11] H. Oh and S. Ha. "A Static Scheduling Heuristic for Heterogeneous Processors," *Proc. Int'l Euro-Par Conf. on Parallel Processing*, pp. 573-577, 1996.

[12] R.J.O. Figuiredo and J.A.B. Fortes. "Impact of Heterogeneity on DSM Performance," *Proc. Sixth Int'l Symposium on High Performance Computer Architecture,* pp. 26-35, 2000.

[13] J. M. Tendler, *et al*., "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5-25, Jan. 2002.

[14] S. Richardson. "MPOC: A Chip Multiprocessor for Embedded Systems." Technical Report HPL-2002-186, Hewlett Packard, 2002.

[15] Digital Equipment Corp. Alpha 21164 Microprocessor Hardware Reference Manual. October, 1996.

[16] K. Olukotun, et al., "The Case for a Single-Chip Multiprocessor," Proc. 7th Int"l Conf. Architectural Support for Programming Languages and Operating Systems, pp. 2-11, 1996.

[17] J. Andrews and C. Polychronopoulos. "An Analytical Approach to Performance/Cost Modeling of Parallel Computers," *J. of Parallel and Distributed Computing*, vol. 12, no. 4, pp. 343-356, 1991.