Staged Tuning: A Hybrid (Compile/Install-time) Technique for Improving Utilization of Performance-asymmetric Multicores

Tyler Sondag
Intel Labs*
tyler.n.sondag@intel.com

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

Abstract

Emerging trends towards performance-asymmetric multicore processors (AMPs) are posing new challenges, because for effective utilization of AMPs, code sections of a program must be assigned to cores such that the resource needs of the code sections closely match the resources available at the assigned core. Computing this assignment can be difficult especially in the presence of unknown or many target AMPs. We observe that finding a mapping between the code segment characteristics and the core characteristics is inexpensive enough, compared to finding a mapping between the code segments and the cores, that it can be deferred until installation-time for more precise decision. We present staged tuning which combines extensive compile time analysis with intelligent binary customization at install-time. Staged tuning is like staged compilation, just for core assignment. Our evaluation shows that staged tuning is effective in improving the utilization of AMPs. We see a 23% speedup over untuned workloads.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Optimization; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.13 [*Software Engineering*]: Reusable Software; D.3.3 [*Programming Languages*]: Language Constructs and Features - Control structures; D.4.1 [*Operating Systems*]: Multiprocessing, Scheduling, Threads; D.4.8 [*Operating Systems*]: Performance - Prediction

General Terms Performance, Measurement

Keywords behavior, performance-asymmetric multicores

1. Introduction

Single-ISA performance-asymmetric multicore processors (AMPs) are a class of multicore processors where all cores support the same instruction set but individual cores may have different characteristics (clock frequency, cache size, in-order vs out-of-order, etc.) [17, 24, 25, 29]. An example would be a 6-core processor where 5 cores operate at 0.8GHz and one at 2.4GHz. AMPs have been shown to be a more efficient alternative to homogeneous multicore processors in terms of overall performance, die space, heat, and power consumption [17, 24, 25, 29]. Further, their potential combination of cores suited for different tasks makes them better suited to a wide range of problems [19, 24].

1.1 The Problems and Their Importance

In general, for effective utilization of AMPs, code sections of a program must be executed on cores such that the resource requirements of a section of code closely matches the resources provided by the target core [25, 34]. To match these resource requirements, both must be known. Figure 1 shows that we must determine what the AMP looks like, how the code segments behave, and which cores are the best fit for each code segment before matching them.

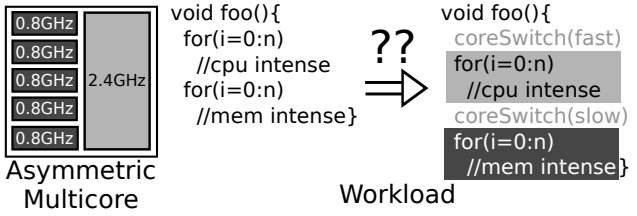

Figure 1. Problem: Given an AMP and a program, how do we determine which code sections execute on which AMP cores?

Programmers may manually perform assignments, however, often this is problematic. This tuning *requires significant expertise* to determine the runtime characteristics of their code as well as the details of the underlying asymmetry. This increases the burden on programmers. This issue is even more troubling for legacy software that are difficult to understand for maintainers tuning them [14].

An automated compile time technique could perform the mapping in some cases. However, portability is a significant challenge when targeting several, possibly unknown, AMPs. For example, suppose the AMP in Figure 1 was unknown. (the same binary likely cannot run on all targets). Another challenge arises when target AMPs are unknown at compile time.

Dynamic techniques have the advantage of being able to observe program behavior on the target AMPs. However, it is difficult to both observe behavior and make decisions during runtime while avoiding excessive overhead, which reduce the observed gains [9].

While each of these options does well in some use cases, we argue that a static automated staged approach can effectively and efficiently optimize programs for a wide range of target AMPs, even when they are unknown at compile time, while putting no additional burden on the programmer and requiring no runtime analysis. We observe that finding a mapping between the program segment characteristics and the core characteristics is inexpensive enough, compared to finding a mapping between the program segments and the cores, that it can be deferred until installation-time for a more precise decision. The extensive static analysis to find program characteristics can be performed once, at compile time, and this data used at low cost for any target AMP. Our approach, staged tuning, performs a one time static analysis of a program at compile time and propagates the results to any target AMP. Then, once the target AMP is known, the program is quickly customized for the AMP using program and machine information previously computed. Staged tuning is like staged compilation [7], just for core assignment.

The benefits of staged tuning include that it is

- *easy to use* – it requires no programmer effort or expertise in program behavior or the underlying AMP

- *portable* – expensive program analysis is done only once per program and communicated to all target AMPs (including those unknown at compile time),
- *efficient* – tuning stages that occur on the target AMP require no additional expensive program analysis,
- *low overhead* – staged tuning requires no runtime monitoring or analysis,
- *faster* – we implemented staged tuning and evaluated it on a physical AMP by comparing to both the stock Linux scheduler (SPEC CPU benchmarks within workloads are on average 23% faster) and closely related work (on average 17% faster), and
- *accurate* – our staged approach creates core assignments finely tuned to each target AMP (we evaluated staged tuning's core assignment for a wide range of potential AMPs and observed greater than 90% accuracy).

We have performed an experimental evaluation of staged tuning for a wide variety of potential AMPs. First, using a physical AMP running in large workloads we see that SPEC CPU benchmarks tuned using our technique show on average 23% speedup over their original binaries. We also demonstrate benefits both in terms of improved performance, reduced overheads, and flexibility over previous work [47]. Furthermore, we see, for a wide range of potential target AMPs, that the key piece of staged tuning, our core assignment, is more than 90% accurate for nearly all targets.

The contributions of this work include the following:

- *Staged tuning*: a novel, fully automatic, compile+install time optimization technique for asymmetric multicore processors (AMPs),
- an implementation of this technique and demonstration of its use on a real AMP,
- an evaluation of staged tuning on a real AMP showing significant improvement over the stock Linux scheduler and over related work [47], and
- a thorough evaluation of the core piece of staged tuning, core assignment, for a wide range of potential AMPs showing more than 90% accuracy.

The rest of this paper is organized as follows. In Section 2 we describe each component of staged tuning in detail. Then, in Section 3 we describe related work. In Section 4 we present our evaluation of the accuracy of staged tuning as well as how it compares to previous work. Finally, Section 6 concludes and discusses future work.

2. Staged Tuning

The goal of staged tuning is to improve the utilization of AMPs, which is important for realizing their potential. We aim to do so while requiring no runtime analysis and no effort from programmers. The key idea behind staged tuning is to perform extensive compile time program analysis but delay core mapping decisions until the target AMP is known thereby automatically creating custom tuned binaries for each target AMP. This also means that the expensive analysis only happens one time and the results are used for each target AMP (even for AMPs that are unknown at compile time). This technique also requires no runtime analysis. Furthermore, this approach puts no additional burden on the programmer.

The core ideas and insights behind staged tuning could be used to detect program segments with similar behavior (demonstrated later). This knowledge could potentially improve a wide range optimizations. Here, we demonstrate how staged tuning is used for a specific optimization, core assignment for AMPs.

Staged tuning consists of three major steps.

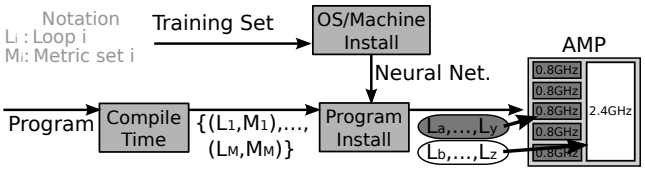

Figure 2. An overview of staged tuning.

First, at compile time, program behavior metrics are computed for the program. These are computed once for each program and are passed with the program to all target machines. Second, at OS (or machine) install time, a neural network is trained for the current AMP. This process is entirely automatic, requires no expertise, is done only once for the entire lifetime of the machine, and is used by all programs executing on the machine. Third, at program install time (i.e., when the program is placed on the target machine) the installer uses the pre-computed metrics and neural network to assign program segments (loops) to types of cores in the AMP. Then, the program is customized to switch cores when it transitions between segments with different core type assignments. Each of these steps is now described in detail.

2.1 Compile Time – Computing Similarity Metrics

At compile time, a static analysis computes approximate metrics of program behavior (e.g., cache behavior, instruction type, etc.).[1] The bottom left of Figure 2 shows that at compile time we take as input a program and output a set of pairs each consisting of a loop and its metric set. Enough metrics must be analyzed such that we can approximate the behavior of each segment on any target machine (with the same ISA). Fortunately, we can afford to spend effort computing as many metrics as desired since this analysis only happens once per program and the *results are used for all target machines*. This analysis may be invoked automatically post-compilation, manually by developers, or by the end-user if no metrics are supplied with the program. In our evaluation, we perform the analysis on binaries after compilation is complete.

The left of Figure 3 illustrates this stage in more detail. Here we have a simple input program with two loops. Static analysis labels these loops and computes a metric set for each. For loop nesting, we compute metric sets for each nesting level separately.

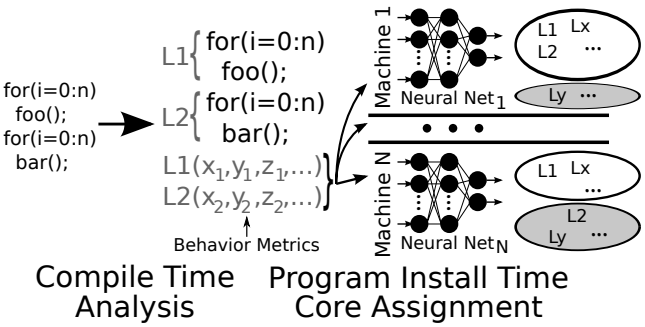

Figure 3. Compile and program install time: At compile time, loops and behavior metrics are determined. This information is sent to each machine with the program. At program install time, a neural network computes core assignment for each loop.

We now give a brief overview of the behavior metrics statically computed for our evaluation. We believe these metrics are a reason-

[1] We avoid using runtime profiles because they require representative inputs which requires additional expertise and programmer effort.

able subset of those necessary to approximate behavior in order to demonstrate the utility of the approach. As we show in Section 4, they work well in practice. Nevertheless, more metrics would most likely be used in practice to give the best possible accuracy.

Instruction Latency The first metric estimates the average instruction execution time. Since we are dealing with a range of target cores all with different implementations of operations (some with in-order and some with out-of-order execution and other issues like memory accesses) we cannot compute how long an instruction will take. However, we can estimate that some instructions will take longer than others (e.g., division takes longer than addition). Thus, the goal becomes estimating some measure of execution time that correlates with behavior on at least most target cores.

We explored several options for this measure including instruction type groups (as in previous work [47]) and estimations derived from monitoring behavior of small programs on modern architectures [18]. The most effective technique we found uses cycle counts on 486 CPUs from an instruction reference [43]. These metric also considers instruction operands (e.g., memory vs. register).

Instruction Cache The next metrics are several rough static estimates of instruction cache behavior. This includes an analysis of both the best and worst cases for simple caches with several levels of associativity (analysis is similar to the work by Ferdinand and Wilhelm [15] on cache behavior prediction). The predicted hit rate for each analysis is used. In future, more sophisticated cache analysis such as [46, 48] can also be incorporated.

Data Cache Approximate data cache behavior is also used. The idea is to statically analyze reuse distances of data accesses [5]. Reuse distance refers to the number of unique accesses between the current accesses and the previous access to the same location. However, since our analysis is static, instead of looking at execution profiles, we analyze instruction operands to compute an approximate best case reuse distance. This reuse distance is then used to place each access into a bucket of similar data accesses (e.g., all accesses with a distance between 2^2 and 2^3 are put in the same bucket). This is similar to part of the MICA technique for clustering whole benchmarks [20]. We use six buckets that collect accesses within a specific range – $[0, 2^0), [2^0, 2^1), \ldots, [2^5, \infty)$. The average (per instruction) number of accesses in each bucket is used.

ILP We include rough static approximations of instruction level parallelism (ILP). Similar to the cache reuse distance analysis, we look at operands for each instruction. Again, we use only static approximations. For each register and memory location accessed by each instruction we perform a backwards search for dependencies. A worst case analysis is used, however, we do not look at some forms of dependence such as blocking of functional units. Like data cache analysis, each instruction is placed into one of five buckets depending on the distance to the previous dependency.

Loop Size The final two metrics are simple and look at loop size in terms of total instruction and basic block count. These metrics on their own do not predict behavior well. However, adding them to the metric set improves accuracy.

2.2 OS/Machine Install Time – Network Training

At some point, staged tuning must map a sets of metric values to types of cores. A goal for staged tuning was to make this process entirely automatic but also take advantage of knowledge about the target AMP. Since some target AMPs are unknown, we chose a machine-learning based approach. Thus, at OS or machine install time, a neural network is trained.[2] This process is entirely automatic, requires no expertise, and is only done once per machine. This neural network is then used for determining the core assignment of program segments for all programs to be optimized for this machine. The top of Figure 2 shows that at OS/machine install time we take as input the training set and output a neural network. Like the output of the compile time stage, the training set consists of pairs of loops and their metrics. To determine the desired network outputs, each loop in the training set is run on the current machine and its behavior observed (we run entire programs but only monitor the loops in the training set). This could be integrated into the OS installation or performed manually after installation. Alternatively, users could download networks computed for similar systems.

A neural network (or artificial neural network) is a model that mimics certain aspects of a biological neural network (i.e., a brain) [38]. This network has layers. The first and last layers are the *input and output layers* respectively. The layers in between are called *hidden layers*. Data is fed into nodes in the input layer. Data is passed through the network where each node takes inputs from the previous layer, multiplies each by some weight, aggregates the result, and passes the result through an *activation function* [38]. To give desired outputs, the network is *trained* using a *training set* consisting of inputs and desired outputs. This training set is repeatedly sent through the network. Each time, weights are adjusted to bring the actual outputs closer to the desired outputs. This continues until a desired error rate is reached.

For our use, the size of the input layer is equal to the number of similarity metrics we have. We have found that networks with three hidden layers, each roughly the same size as the input layer, work best. While we observed that periodic functions (e.g., sin, cos) help achieve the desired error rate most quickly, the output networks were much less accurate than networks using the sigmoid logistic function or similar approximations. The size of the output layer is equal to the number of desired core assignment "groups". In this work, we have one group corresponding to each type of core in the AMP. More groups could be used (e.g., one for program segments that don't have a core preference), but we have not explored that.

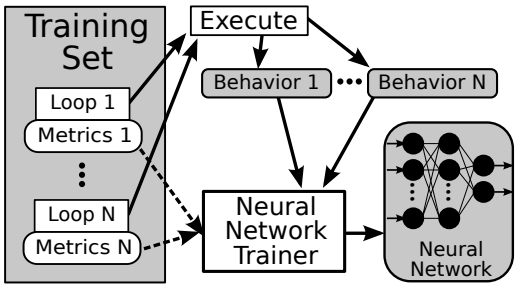

Figure 4. OS install time. To determine desired outputs, loops in the training set are executed and behavior is gathered.

For example, in Figure 4 the network has two output nodes. Thus, loops will be divided into two groups, each corresponding to a type of core in the AMP. Each node in the output layer computes a probability that the input belongs to the corresponding group (should execute on the corresponding type of core). So, if we have two output nodes, n_1 and n_2 we say the input belongs to group c_1 if $out(n_1) > out(n_2)$ where out gives a node's output value.

Our training set consists of a set of benchmarks (we use the SPEC CPU benchmarks, however any set of programs could be used or crafted for this purpose) and precomputed metrics for the loops in each benchmark (shown on the left of Figure 4). Each loop is a single training example in our training set. These metrics for each loop make up the inputs of each member of the training set. Since the outputs are probabilities for which type of core the loop should be assigned to, we need a way to determine, for each loop in our training set, which type of core is ideal.

[2] Another form of machine learning could potentially be used instead, neural networks are simply the technique we chose due to prior experience.

To determine the desired core assignment, execution behavior is monitored for each loop (top right of Figure 4). To do so, each loop is instrumented to gather performance data. We monitor instructions (actually μops) per cycle, however, other metrics could easily be used if desired. After execution, the behavior of all executions (e.g., nested loops that executed many times) are averaged. This is done once for each type of core. Suppose the performance data gathered was IPC (instructions per cycle) and that two types of cores exist. Then, a single threshold of "IPC change" is used. That is, any loop whose IPC difference when executed on the two core types is above this threshold goes in one group (i.e., is assigned to the core giving higher IPC). The other loops go in the other group (i.e., the core giving lower IPC). The idea is that the most computationally intensive loops should execute on the fastest cores while others (perhaps memory intensive loops) could save resources by executing on the slower cores (and see less slowdown than the computationally intense loops). For more types of cores, more complex criteria would be used (e.g., multiple thresholds).

To avoid over-fitting, values that lie close to, within ϵ, from the threshold(s) are set with equal probability for the groups (core types) on either side of the threshold. For example, if we have a threshold of 0.5 and two loops with IPCs of 0.49 and 0.51, we want these loops to be considered similar (and they likely have similar metric values). Informing the neural network otherwise may hurt accuracy. In our experiments, ϵ ranges from 0.01 when cores only differ in frequency to 0.03 for cores with significant differences. These values avoid over-fitting loops that lie directly on either side of the threshold while at the same time avoid capturing too many loops as being fit for either group (core type).

Once we have determined the desired output for each loop, the network is trained (bottom right of Figure 4). This entire step is lengthy due to gathering the actual behavior of each loop. Fortunately, this step only needs to be done one time, when the machine is set up. Throughout the entire lifetime of the machine, the same neural network may be used. Additionally, it would be reasonable that users could obtain a pre-trained network based on their machine configuration.

We can influence the core assignment distribution through the training set (by adjusting the threshold used to determine the desired outputs). This means the distribution of loops assigned to each core type can be tuned based on the distribution of core types in the target AMP and/or the desired behavior. For example, suppose we have an AMP with two core types (e.g., fast and power efficient). If we want efficient power use, we may only put segments in the "fast" core's group when the benefits of doing so are large.

2.3 Program Install Time – Core Assignment

The final stage, core assignment, happens at program install time (when the program is placed on the target machine). This stage is automatic and only happens once per program, per target machine. The bottom right of Figure 2 shows that loops and their metrics along with the neural network are input. The output of this stage is a modified program where loops are assigned to cores in the AMP. This process can be built into install scripts or done by end-users.

This stage is illustrated in more detail on the right of Figure 3. During this stage, the metrics for the program (computed at compile time) are submitted to the neural network (trained at OS install time) to compute the core assignment. A benefit is that, since all expensive analysis is completed before this stage, core assignments are computed very quickly (on average < 0.025s). This stage also only occurs once per program statically (at install time).

For dealing with loop nesting, we determine an assignment for each nesting level. Then, these assignments are merged if deemed appropriate. For example, if an outer loop has the same assignment as its inner loop, we ignore the assignment for the inner loop. If the assignments are different (or if the outer loop contains many inner loops) we use heuristics (strength of the core assignment) to choose if one of the assignments or both will be used.

Finally, the program is modified to include the core switching code based on the computed core assignment. When the program is run, it takes advantage of the core assignment with no runtime monitoring or analysis overhead.

3. Closely Related Work

Prior to quantitatively comparing staged tuning with the closest related work, we qualitatively compare and contrast with closely related ideas.

Sondag and Rajan proposed phased-based tuning, a hybrid (static and dynamic) analysis for improving utilization of AMPs [44, 45, 47]. Their technique exploits similarity between code segments to reduce dynamic analysis overhead. A static analysis partitions program segments into groups of similarly behaved segments. Then, dynamic analysis is assigns entire groups to cores of the AMP based on the observed behavior of a sample of program segments in the group. Staged tuning also statically "groups" program segments that we believe will behave similarly. However, instead of computing core assignments at runtime, we do so at program install time to further decrease runtime overhead (and remove dynamic analysis overhead).

Becchi *et al.* [3] propose a dynamic assignment technique making use of the instructions per cycle (IPC) of program segments. Their work focuses on the load balance across cores whereas we aim to maximize throughput. Shelepov *et al.* [9] propose a technique which does not require dynamic monitoring (uses static performance estimates). However, this technique assigns cores at a per benchmark granularity and thus does not consider program phase behavior. Li *et al.* [25] and Koufaty *et al.* [23] focus on load balancing in the OS scheduler. They modify the OS scheduler based on the asymmetry of the cores. While this produces an efficient system, the scheduler needs knowledge of the underlying architecture. Our work differs from these in the following way. First, this work is not directly concerned with load balancing. Second, this work focuses on scheduling the different phases of a program's behavior.

Tam *et al.* [49] determine core assignments based on increasing cache sharing. They use cycles per instruction as a metric to improve sharing for symmetric multicores. Kumar *et al.* propose a temporal dynamic approach [33]. After pre-defined time intervals, a sampling phase is triggered. After this phase, the system makes assignment decisions for all processes. After a fixed period of time, this procedure repeats (throughout the program's entire execution). Staged tuning differs in that it does not require runtime monitoring.

Cao *et al.* [6] studied using AMPs for virtual machine (VM) services. They found that small/simple cores are well suited for VM services since VM code is often asynchronous, parallel, non-critical, and poorly utilizes larger more complex cores. Our work differs in that we target code from non-managed languages. As part of future work, it would be interesting to explore how staged tuning behaves for VM services.

Other related works focus on providing and choosing from multiple variants of code segments for heterogeneous systems. Sandrieser *et al.* [41] proposed using "explicit platform descriptions", which allow "mainstream" programmers to implement a single annotated program that experts later tune for a heterogeneous system (runtime systems may pick between implementation variants). Benkner *et al.* [39] propose a component model for providing multiple versions of performance-critical code parts of applications and using various techniques to select the most suitable version for the given heterogeneous system. Staged tuning differs in that it is automatic and focuses on single-ISA AMPs. Jiménez *et al.* developed a scheduler aimed at making use of heterogeneity in terms of CPU

and GPU [51]. Their work is targeted at a specific type of heterogeneous multicores with different ISAs whereas our work focuses on single ISA performance asymmetric multicores. Since they focus on cores with different ISAs they require the programmer to note which functions can be executed on both core types or provide implementations for both core types.

Among this work, we find phase-based tuning [47] most similar and thus choose it for our quantitative comparisons.

4. Evaluation

Our hypothesis is that staged tuning improves utilization of AMPs over the stock Linux scheduler and the most similar prior work [47]. We also hypothesize that its key piece, core assignment, is accurate for a wide range of potential AMPs.

To evaluate these hypotheses, we first compare staged tuning to the stock Linux scheduler on a physical AMP. We also compare to a similar technique, phase-based tuning [47]. Next, we evaluate the main component of our technique, core assignment, for a wide range of potential AMPs.

4.1 Staged Tuning for a Physical AMP

Research Question 1: *Does staged tuning improve performance on an AMP over the stock Linux scheduler?*

We now briefly describe the experimental setup for evaluating this question and then discuss the observed benefits.

4.1.1 Experimental Setup

Machine setup We created a physical AMP using the six core Opteron 2431 with five cores frequency scaled down to 0.80GHz and one at 2.40GHz. We chose this system because of all our options, the frequency range is the largest ("fast" core frequency is 3x "slow" core) and the core type ratio is realistic (highly skewed). While this AMP may not be as ideal as something that would be produced for this purpose, we believe it is the best AMP we could construct with our resources without resorting to simulation.

Workload setup Workloads are constructed as follows. For a workload of size n we have n benchmarks running simultaneously at all times. In our experiments, $n = 24$. When a benchmark completes, another randomly selected benchmark is started. The goal is to maintain a fixed number of running benchmarks. Workloads run for a fixed amount of time, in this case 60 minutes. We capture the selected benchmarks so that the same experiments can be replayed for different configurations (i.e., Linux scheduler vs. staged tuning). This is similar to previous work [47].

Throughout these experiments, the benchmarks share (or compete for) the different types of cores. Thus, if any benchmark is greedy (takes a "fast" core when other more suited benchmarks are competing for it) overall throughput will suffer. Note that we do not introduce parallelism or directly enforce that all types of cores are kept busy (though indirectly this is usually the case). However, these topics would be interesting to explore in future work.

The benchmarks we used are those from the SPEC CPU 2000 suite[3]. This suite is used instead of more recent suites (e.g., SPEC CPU 2006) since it runs in a reasonable amount of time under fine grained monitoring (necessary for determining ideal assignment) on our older machines.

Neural Network Training We use the FANN library [32] for constructing and training our neural networks. In our experiments, we compute a grouping (i.e., core assignment) for each individual benchmark. To do so, we use a technique called leave-one-out cross-validation [38, pp.663]. That is, when computing a grouping for a benchmark, we train a new neural network with the rest of the benchmarks (i.e., we exclude the current benchmark and all its loops from the training set – also excluding all other runs under different input sets). In this way, *the neural networks used for our experiments for each benchmark are not trained with the benchmark being analyzed*. In practice, the training set could include a more wide range of program types.

The ideal neural network outputs for the training set are determined as follows. First, a threshold IPC difference is determined such that if a segment's IPC difference between the core types is below the threshold it will achieve enough efficiency on the faster core to justify taking the space on the single faster core. Loops with IPC differences on either side of this threshold are assigned to the different types of cores.

4.1.2 Performance of Staged Tuning

Runtime Overheads Additional space is required in the binary for the dynamic optimization code (after installation, storage of behavior metrics is not necessary). Staged tuning only requires on average 2.95% space overhead.

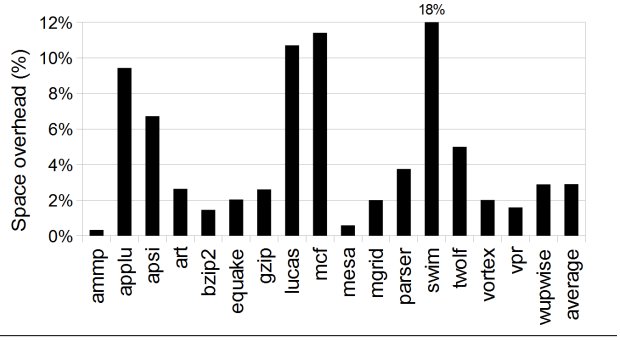

Figure 5. Space overhead of optimization code.

Figure 5 shows the runtime space overhead of staged tuning per benchmark. The figure shows that nearly all benchmarks have less than 4% space overhead (most are around 2%). A few benchmarks have a significantly higher space overhead. Some of these are small programs (e.g., swim and mcf) whereas the others have many non-nested loops (e.g., applu and lucas).

Extra time is spent executing dynamic optimization code, this is time overhead. To measure this, we set all core switches to "all cores" (new code is executed and makes the same library calls but no core switches occur). Figure 6 shows the time overheads of individual benchmarks running in isolation.

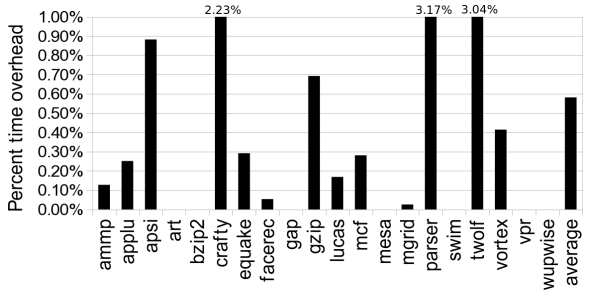

Figure 6. Time overhead of dynamic optimization code.

We see that most benchmarks have extremely low time overheads even when running in isolation. Most benchmarks have below 0.3% time overhead and several have no noticeable overhead.

[3] excluding perlbmk, gcc, eon, fma3d, and sixtrack which either do not execute or analyze properly

Static Overhead Sending compile time computed similarity metrics with programs has some space overhead. Approximately 80 bytes per loop is needed in our current implementation, however, this could easily be reduced. Figure 7 shows the required space overhead for each benchmark.

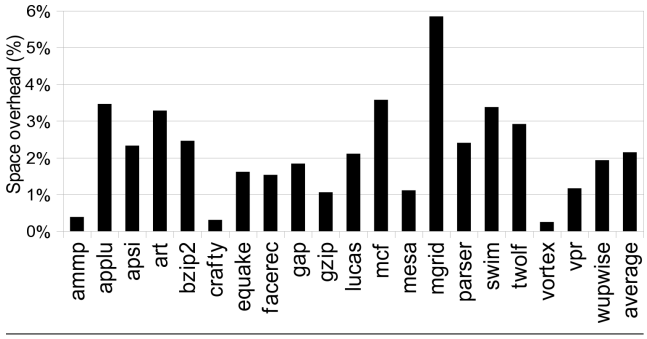

Figure 7. Space overhead of behavior metrics.

The average space overhead is approximately 2%. This could be reduced by using more compact data types (e.g., we use, but do not need, four bytes to store the basic block count for loops). Note that this overhead does not necessarily impact the final binary size (metrics storage is only required until core assignment is completed).

Install time core assignment also has a cost, even though it is only done once for each program. We observe, for the benchmarks studied, that staged tuning takes less than 0.03 seconds at program install time to determine the core assignment. Additionally, the current implementation of this phase is not yet optimized for fast execution. Thus, we expect this time to reduce further with a fine-tuned implementation.

Average Process Speedup To measure how staged tuning improves program speed, we consider average process speedup for processes in our workloads. Using staged tuning, we observe approximately 23% average speedup over the stock Linux scheduler.

Figure 8 shows the average speedup per benchmark (for benchmarks that complete under stock scheduling). Many benchmarks

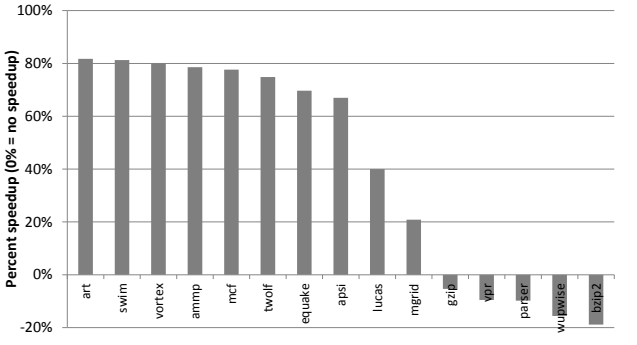

Figure 8. Per-benchmark staged tuning speedup over the stock Linux scheduler.

see large average speedups whereas only a few observed small slowdowns. Those with slowdown either tend to prefer slower cores for large chunks of execution, switch cores frequently (aside from switching overhead this can hurt cache behavior), or spend a large part of their runtime executing on cores with high contention.

Fairness Of course, speedup doesn't take into account starvation and other scheduling issues. That is, average speedup could come at the cost of some processes starving. Previous work [47] considered fairness using two metrics: max-flow and max-stretch (as defined by Bender *et al.* [4]). Max-flow can be thought of as the process with the longest execution time (from arrival to completion time). The use of staged tuning reduces max-flow (lower is better) by 17% over the stock Linux scheduler. Max-stretch can be thought of as the largest slowdown for an individual process. That is, if any individual process slows down, this metric will increase. With staged tuning, max-stretch is reduced by 28% (lower is better) over the stock Linux scheduler.

Figure 9 shows the average max-stretch decrease per benchmark (for all benchmarks that complete under stock scheduling).

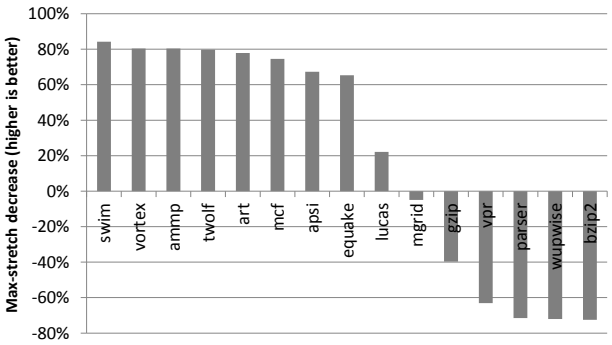

Figure 9. Per benchmark max-stretch decrease given by staged tuning compared to stock Linux scheduler (higher is better).

Like speedup, many benchmarks see big gains in fairness, however, several see decreased fairness. These benchmarks for which fairness got worse match up with those where speedup was small (or there was a slowdown). This is expected since, for example, even though those benchmarks that often prefer the "slower cores" see less slowdown compared to program segments preferring the "faster cores", they still see slower execution on the "slower cores" than the "faster cores".

4.1.3 Comparison to Phase-based Tuning

Research Question 2: *Does staged tuning outperform phase-based tuning (closely related prior work described in Section 3)?*

Experimental Setup We compare both techniques on our AMP described above (5x0.8GHz, 1x2.4GHz). We chose this AMP over one similar to previous work [47] since the core types vary more and the core type ratio is more realistic. Because there are fewer "fast" cores than in previous work (1 of 6 rather than 2 of 4), we expect speedups to be smaller than those previously reported [47].

For phase-based tuning, we manually tune the metric set and its weights (through a combination of manual, statistical, and automated search) such that we create two groups with approximately 85% accuracy (the same number of groups and accuracy as in previous work [47]). Workloads constructed as described above.

Overhead Comparison For space overhead (i.e., space required for dynamic analysis and optimization code), staged tuning gives an average decrease of 25.3% over phase-based tuning (3.98% vs. 2.95%) for the best phase-based tuning technique [47, pp.7]. This reduction is because the inserted code no longer contains code for monitoring behavior and making core mapping decisions.

For time overhead (i.e., dynamic analysis and optimization), staged tuning gives an average decrease of 55.9% over phase-based tuning (0.17% vs. 0.075%). This is measured (as in previous work [47]) on the runtimes of entire workloads instrumented to switch to "all cores". This differs from our earlier results that measured overheads for programs running in isolation. This reduction is because the dynamic code no longer checks for dynamic core mapping decisions.

Speedup Comparison Using phase-based tuning, under the same parameters as previous work [47] (excluding target AMP and metric set which is tuned for our new AMP), we see approximately 6% average process speedup over the stock Linux scheduler. Using staged tuning, we observe approximately 23% average speedup.

This improvement is due to three factors. First, staged tuning has a core assignment distribution more similar to the core type distribution (this is also why we see a lower average process speedup than previous work observed on a different, more balanced AMP). Second, we have zero runtime monitoring and analysis overhead (and no "warm-up phase" when core assignment is not yet known) whereas phase-based tuning performs runtime monitoring. Third, staged tuning's core assignment is more accurate (a detailed comparison is shown in our companion technical report).

Fairness Staged tuning reduces max-flow by an additional 5% (17% decrease vs. 12% with phase-based tuning). Max-stretch is 8% lower with staged tuning (28% decrease instead of 20%).

4.2 Core Assignment Accuracy

Research Question 3: *Is they key piece of staged tuning, its core assignment, accurate for a wide range of AMPs?*

To answer this question, we first describe our experimental setup for approximating a wide range of potential AMPs. Then, we demonstrate the accuracy of our core assignment for these AMPs.

4.2.1 Experimental setup

We now describe our experimental setup including how we determine behavior for target configurations, hardware and software setups, and techniques used to measure accuracy.

System Setup All systems used for experiments are physical systems running GNU/Linux. Figure 10 shows all core types used in our experimentation. We include modern processors (e.g., Core i7, Opteron 6168), power efficient processors (e.g., Atom, Pentium M), older processors (e.g., Pentium 4), and some in between. For each core type, frequency was varied (if possible) as well creating a wide range of core types (in total we tested 22 core types).

Series and Model	Frequencies (GHz)	L1 (i/d) (KB)	L2 (KB)	L3 (MB)	Cores
Core i7, 870	1.2,1.6,2.0,2.4[4],2.93	32/32	4x256	8	4/8
Atom, N270	0.8,1.6	32/24	512		1/2[5]
Core 2 Quad, Q6600	1.6,2.4	32/32	2x4096		4
Core 2 Duo, E6300	1.6,1.83	32/32	2048		2
Pentium 4, 2.0	2.0	12K[6]/8	512		1
Pentium M, 725	0.8,1.6	32/32	2048		1
Opteron, 2431	0.8,1.2,1.5,1.9,2.4	64/64	6x512	6	6
Opteron, 6168	0.8,1.3,1.9	64/64	6x512	12	12

Figure 10. Core types used. All L1 caches are private and split (instruction + data). For cores with hyperthreading, the number physical cores and total threads are shown.

Computing Actual Behavior Behavior is gathered using PAPI [22]. Our own program analysis and instrumentation framework, which works on binaries, detects loops (using standard algorithms [30]) and inserts PAPI calls. All machines run the same binaries. When sufficient information is unavailable to safely instrument or detect a loop (e.g., unknown branch target), it is ignored. For loops executed multiple times, the average behavior of all executions is taken.

We consider both the behavior of loops running on an individual core (i.e., the IPC of the loop when run on a specific core) and difference in behavior between two core types (i.e., difference in observed IPC when run on two different core types) in asymmetric configuration (as used by phase-based tuning [47]). For asymmetric configurations, we approximate difference in behavior using two core types, sometimes in different systems. While some of these AMPs do not exist and other characteristics of each machine may differ, we believe that the difference in behavior is representative of some potential AMP. We focus on asymmetries with two core types because previous work suggests that two types are sufficient to realize the benefits of AMPs [19, 24].

Accuracy Metrics To determine the accuracy of core assignment, we use several techniques.

Raw Accuracy Raw accuracy is the percent of loops that "fit" in their group (i.e., fit on the corresponding core). If we have two groups both containing five loops (10 loops total) and all are "correctly" grouped except one (ideal grouping is determined the same way training sets are created), then this grouping has 90% raw accuracy.

Group Accuracy Suppose we have two groups, one with eight loops, the other with two. The group with eight loops is completely accurate (100%) whereas the other is entirely incorrect (0%). In this case, the group accuracy is 50% (i.e., each group gets equal weight regardless of its size). This case would be reported as 80% accurate with the raw accuracy metric. This case could occur if we were to choose an extreme threshold and cause the size of one group to become very small. Simply placing all loops in the larger group will give a high raw accuracy. Group accuracy solves this problem by giving a 50% group accuracy.

p-value We consider the p-value for the statistical test where the null hypothesis is that the two groups have the same mean. A sufficiently low p-value will reject this hypothesis. That is, the two groups have a statistically significant difference in their means.

Boxplots Finally, we consider a visual analysis using boxplots, which help illustrate the distribution of loop behavior in each group. Consider the left of Figure 12. Here, we used staged tuning to divide loops into two groups, a plot is shown for each. The boxes represent the inner quartile range of the loop behavior for the group. The lines on either end extend to the minimum and maximum of the behavior in the group (excluding outliers, shown as circles). The line in the middle of the box shows the median behavior. For our purposes, ideally there would be no overlap between groups (i.e., everything assigned to a core type would be either above or below some threshold). We believe a good grouping should at least not overlap the inner quartiles (i.e., boxes).

4.2.2 Core Assignment Accuracy

We now evaluate the accuracy of staged tuning for a variety of asymmetric configurations. To do so, we run benchmarks on a wide variety of machines and observe their behavior. This behavior is used to assign loops to cores giving us an ideal assignment. We then compare the ideal assignment with staged tuning's assignment for each configuration along four dimensions: raw accuracy, group accuracy, statistical test p-value, and visual analysis.

Some configurations were chosen because they combine high power and power saving cores (e.g., i7 with Atom or Pentium M). Others were chosen that have similar clock frequency but differ in other (known or unknown) ways (e.g., Core 2 Quad and Opteron). Finally, in configurations cores differ only in frequency.

Figure 11 gives the average raw and group accuracy (defined in Section 4.2.1) of staged tuning for each configuration. For all, we see a p-value of < 0.01. Thus, we leave this p-value out of the table. We focus on two groups since this is a likely use case (e.g., assign to a fast or slow core).

[4] Actually 2.39GHz, but rounded to avoid confusion with 2.93GHz.

[5] Uses in-order execution.

[6] Uses a 12K μop trace cache – similar to an 8-16KB i-cache [40].

Core Type 1		Core Type 2			Group
Model	Freq.	Model	Freq.	Accur.	Accur.
i7	2.93	i7	1.20	95.0%	91.0%
*i7	2.93	Atom	0.80	94.0%	93.6%
i7	2.40	Core 2 Quad	2.40	94.5%	92.3%
i7	1.60	Core 2 Quad	1.60	94.5%	88.8%
i7	2.93	Core 2 Duo	1.60	98.7%	90.0%
i7	1.60	Core 2 Duo	1.60	95.3%	92.5%
i7	2.93	Pentium 4	2.00	92.7%	92.4%
*i7	2.93	Pentium M	0.80	91.6%	91.4%
i7	2.40	Opteron 2431	2.40	92.6%	89.0%
i7	2.93	Opteron 2431	0.80	91.0%	88.9%
Atom	1.60	Atom	0.80	92.3%	89.2%
Atom	1.60	Core 2 Quad	1.60	93.1%	93.4%
Atom	1.60	Core 2 Duo	1.60	92.8%	90.2%
Atom	0.80	Core 2 Duo	1.60	93.2%	92.0%
Atom	0.80	Opteron 2431	2.40	92.5%	88.0%
Atom	0.80	Opteron 2431	0.80	96.3%	90.1%
Core 2 Quad	1.60	Core 2 Duo	1.60	94.3%	94.3%
Core 2 Quad	2.40	Pentium 4	2.00	94.4%	92.6%
Core 2 Quad	2.40	Pentium M	0.80	94.3%	93.3%
Core 2 Quad	2.40	Opteron 2431	2.40	96.0%	94.8%
Core 2 Quad	2.40	Opteron 2431	0.80	95.8%	93.8%
Core 2 Duo	1.83	Pentium 4	2.00	95.2%	93.4%
Core 2 Duo	1.83	Pentium M	0.80	94.5%	92.5%
Pentium 4	2.00	Pentium M	0.80	95.2%	95.5%
Pentium 4	2.00	Opteron 2431	2.40	94.0%	91.7%
Pentium 4	2.00	Opteron 2431	0.80	92.8%	90.9%
Pentium M	0.80	Opteron 2431	2.40	90.2%	91.3%
Pentium M	0.80	Opteron 2431	0.80	94.2%	91.1%
Opteron 2431	1.90	Opteron 6168	1.90	97.8%	95.8%
Opteron 2431	0.80	Opteron 6168	0.80	97.6%	94.9%

Figure 11. Accuracy per configuration. Configurations marked with * have boxplots shown later.

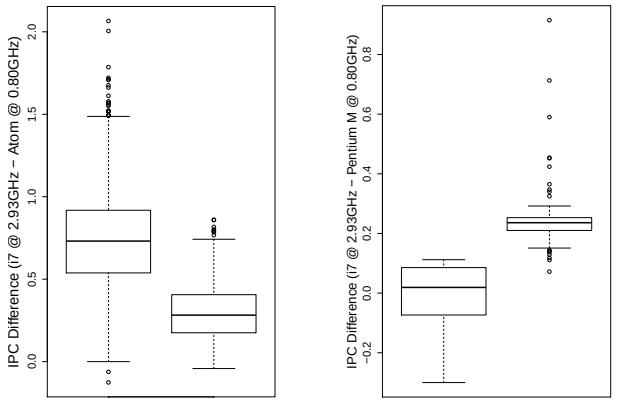

Figure 12. Core assignment behavior distribution across all benchmarks. For all configurations, we see a clear difference in behavior between loops assigned to different cores.

The table shows that for nearly all configurations, we see greater than 90% raw and group accuracy. For selected configurations, Figure 12 contains boxplots that show the distribution of the behavior of groups across all benchmarks. One configuration uses our two most varied core types, the i7 and Atom at their most extreme frequencies. We also show a combination of the i7 and Pentium M running at their extreme frequencies. This differs from the previous configuration in that both cores do out-of-order execution.

For all configurations, we see a significant difference in the behaviors for each group. Finally, for our most varied configuration (i7 @ 2.93GHz with Atom @ 0.80GHz) we present the boxplots for the grouping for each benchmark for each input set in Figure 13.

Nearly all benchmark and input combinations show a significant difference in the distribution of behavior for the two groups. Two benchmarks, ammp and gzip, differ in that they only give a single group (in both cases, the second group). Fortunately, as we can see, the behavior of all loops in these benchmarks fit nicely into this group. The only benchmark and input combination where the inner quartile ranges slightly overlap is bzip2 with its third input (denoted bzip2.2 in the figure). Fortunately, the overlap is small, and the accuracy for this grouping is still 86%.

4.2.3 Single Core Type Grouping

We also adapt the core assignment technique to group loops based on their predicted behavior on a single core. That is, instead of predicting ideal core assignment based on IPC difference between two core types, we predict which loops will have similar behavior (for our tests, IPC) on some individual core type. For example, if a loop has an IPC above some threshold it should be placed in one group, while all other loops are placed in another group.

Core Type		Num.	Raw	Group
Model	Freq.	Groups	Accur.	Accur.
*i7	2.93	2	92.2%	94.0%
i7	2.93	3	85.0%	89.3%
Atom	1.60	2	88.4%	87.3%
*Atom	1.60	3	89.8%	88.1%
Core 2 Quad	2.40	2	94.0%	93.9%
Pentium 4	2.00	2	98.3%	96.6%
Pentium M	1.60	2	92.2%	92.2%
Opteron 2431	2.40	2	92.8%	92.5%

Figure 14. Accuracy of behavior groups for various core types. Cores marked with * have boxplots shown later.

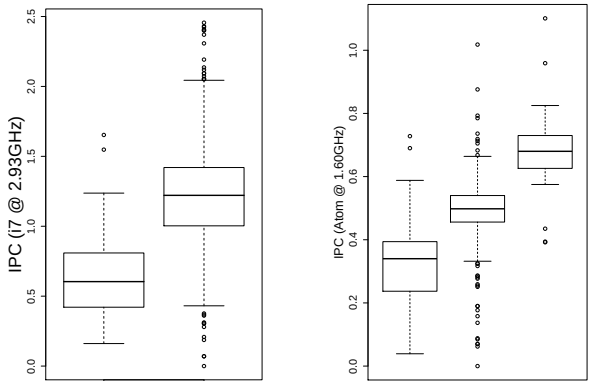

Figure 15. Single core behavior group distribution across all benchmarks. We see a clear difference in group behavior.

Figure 14 gives the average raw and group accuracy (defined in Section 4.2.1) when staged tuning's core assignment technique is adapted to group programs segments with similar behavior. For all, we see a p-value of < 0.01. Thus, we again leave this p-value out of the table. We also include a few experiments with three groups to demonstrate the flexibility of the behavior prediction.

Again, we see above 90% accuracy for most cores. For selected configurations we present boxplots in Figure 15 that show the distribution of the behavior across all benchmarks. We show plots for our most complex (i7) and simple (Atom) cores using both two and three groups. Again, we see clear differences in group behavior.

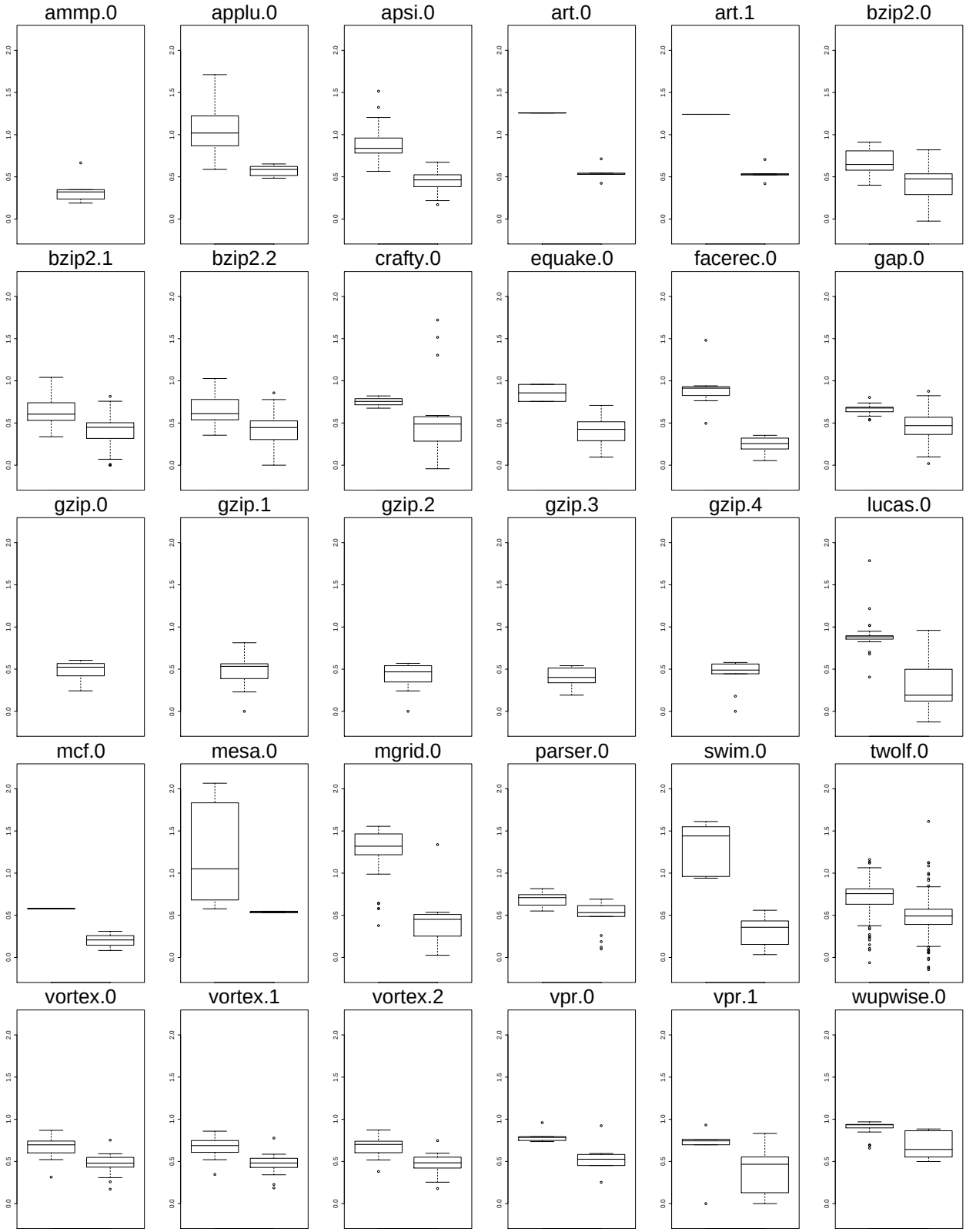

Figure 13. Boxplots for each benchmark for the configuration containing the two cores: Intel i7 @ 2.93GHz and Intel Atom @ 0.80GHz. Behavior is the difference between the two core types. Numbering after the benchmark name denotes input set.

5. Other Related Work

Sherwood *et al.* [42] developed a technique for clustering segments of execution. Staged tuning differs in that we group structural elements of the program rather than execution time. Further, their clustering is based on profiles of executed basic blocks rather than our statically computed behavior metrics.

Choi and Yueng [8] use machine learning to determine efficient thread distributions on SMT systems. Periodically throughout execution threads are reassigned. We analyze behavior based on program structure rather than time. Also, they train their network dynamically whereas we train and use our network statically.

AbouGhazaleh *et al.* [31] use machine learning to dynamically scale clock frequency in embedded systems. Representative inputs are used to determine efficient power management policies, which are used for training. Their compiler uses this information to optimize the program. At runtime, by monitoring performance periodically, decisions are made regarding the power management policy. Our "phases" are determined without ever running the program.

Wang and O'Boyle [52] use machine learning to help partition stream-based programs onto cores. The goal is to choose a good combination of parameters (e.g., level of loop unrolling, number of threads per loop, when to split and join, etc). Unlike their approach, we do not consider modifications to program structure. Further, we focus on behavior of individual segments rather than workloads.

Dubach *et al.* [12] use machine learning to dynamically predict desired hardware configurations for program phases. Our technique does not determine the best configuration for phases, instead, it chooses the best from a set of choices.

Hoste and Eeckhout [20] use execution profiles to cluster benchmarks with similar behavior. Unlike their work, all of our analysis and grouping is static. Also, we assign program segments rather than whole programs.

Agakov *et al.* [13], Püschel *et al.* [28], and Ganapathi *et al.* [16] use machine learning to focus the search of the space of candidate optimizations. The idea is that searching, applying, and evaluating the optimization space is time consuming. Further, the best optimization set is system dependent. They show that machine learning can improve this process by intelligently searching the optimization space. Our approach is similar in that we use machine learning to handle target architecture details. However, we focus on predicting approximate program behavior rather than searching candidate program transformations.

Related work also dynamically uses machine learning and profiling to determine the best algorithm, optimization, or implementation for a given target machine [1, 10, 11, 21, 26]. Similarly, de Mesmay *et al.* [11] propose adapting applications (e.g., picking the best algorithm for the given input on the target machine) at installation time rather than at runtime. The idea is to determine the best adaptation for different problem (input) sizes. Then, at runtime, no adaptation is needed since the application knows the best version of the application to use. We also use an install time approach to reduce dynamic overhead. Our work differs in that we do not require execution or runtime profiles of the candidate program to perform our core assignment.

6. Conclusion

Asymmetric multicore processors are an important class of multicore processors. However, techniques for their efficient use, especially in the presence of a wide range of potential AMPs, are still needed. In this work, we have developed staged tuning, a novel technique for effectively utilizing AMPs. The key idea is to perform extensive compile time program analysis but delay core mapping decisions until the target AMP is known. On a real AMP, we observe a 23% speedup when using staged tuning compared to untuned versions of the same programs. Compared to a similar previous work which creates a single binary for all machines we see an additional 17% average program speedup. Our results have shown that we can, at greater than 90% accuracy on average, statically assign program segments to cores for a wide range of target AMPs.

Future work includes exploring tighter integration with the OS scheduler and investigating the applicability of the key ideas behind our core assignment technique for other problems. We have shown that our technique can accurately group program segments that are likely to have similar runtime behavior on a target core. Thus, one direction to explore is finding other (static or dynamic) optimizations that can make use of this behavior information to either make better decisions statically or reduce dynamic analysis overhead. We would also like to explore whether Staged Tuning is a better fit with some of our recent work on implicitly concurrent programming languages [2, 27, 35–37, 50]. Goal of these languages is to hide the details of concurrency and platforms from the programmer. Staged tuning serves to hide the details of the AMP from the programmer, so it could contribute.

References

[1] A. Tiwari *et al.* Auto-tuning full applications: A case study. *Int. J. High Perform. Comput. Appl.*, 25(3), 2011.

[2] M. Bagherzadeh and H. Rajan. Panini: a concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th international conference on Modularity (Modularity'15)*, pages 93–108. ACM, New York, NY, USA., 2015.

[3] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *International Conference on Computing Frontiers*, 2006.

[4] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Annual Symposium on Discrete algorithms*, 1998.

[5] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001.

[6] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, 2012.

[7] C. Chambers. Staged compilation. In *PEPM '02*, 2002.

[8] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *ISCA*, 2006.

[9] D. Shelepov *et al.* Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[10] A. Danylenko, C. Kessler, and W. Löwe. Comparing machine learning approaches for context-aware composition. In *Proceedings of the 10th international conference on Software composition*, 2011.

[11] F. de Mesmay, Y. Voronenko, and M. Püschel. Offline library adaptation using automatically generated heuristics. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.

[12] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control, 2010.

[13] F. Agakov *et al.* Using machine learning to focus iterative optimization. In *CGO*, 2006.

[14] H. Fahmy and R. C. Holt. Software architecture transformations. In *ICSM*, 2000.

[15] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.*, 17:131–181, December 1999.

[16] A. Ganapathi, K. Datta, O. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, 2009.

[17] M. Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric cores. *Tech. Report - Intel*, 2008.

[18] T. Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, March 2011. http://gmplib.org/~tege/ x86-timing.pdf.

[19] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *ICCD*, 2004.

[20] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *Micro, IEEE*, 27(3), 2007.

[21] J. Ansel *et al.* Petabricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

[22] J. Dongarra *et al.* Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD Workshop*, 2003.

[23] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, 2010.

[24] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 2005.

[25] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC*, pages 1–11, 2007.

[26] X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *CGO*, 2004.

[27] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE '10: Ninth International Conference on Generative Programming and Component Engineering*, pages 63–72. ACM, 2010.

[28] M. Püschel *et al.* SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 2005.

[29] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, 2008.

[30] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Academic Press, 1997.

[31] N. AbouGhazaleh *et al.* Integrated CPU and L2 cache voltage scaling using machine learning. In *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2007.

[32] S. Nissen. Implementation of a fast artificial neural network library (FANN). Technical report, Dep. of Comp. Sci. University of Copenhagen, 2003. http://fann.sf.net.

[33] R. Kumar *et al.* Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, page 64, 2004.

[34] R. Kumar *et al.* Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, 2006.

[35] H. Rajan. Building scalable software systems in the multicore era. In *Proceedings of the 2010 FSE/SDP workshop on Future of software engineering research (FoSER'10)*, pages 293–298. ACM, 2010.

[36] H. Rajan. Capsule-oriented programming. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[37] H. Rajan, S. M. Kautz, and W. Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (ONWARD'10)*, pages 790–805. ACM, New York, NY, USA,, 2010.

[38] S. Russell and P. Norvig. *Artificial intelligence: a modern approach, 2nd ed.* Prentice Hall, 2010.

[39] S. Benkner *et al.* Peppher: Efficient and productive usage of hybrid computing systems. *Micro, IEEE*, 31(5):28 –41, 2011.

[40] D. Sager. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[41] M. Sandrieser, S. Benkner, and S. Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *IWMSE*, 2011.

[42] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, pages 45–57, 2002.

[43] Z. Smith. The intel 8086 / 8088/ 80186 / 80286 / 80386 / 80486 instruction set, September 2011. http://80386.tk/.

[44] T. Sondag, V. Krishnamurthy, and H. Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *Proceedings of the 4th workshop on Programming languages and operating systems*, page 7. ACM, 2007.

[45] T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 73–80. IEEE Computer Society, 2009.

[46] T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *RTSS '10: The 31st IEEE Real Time Systems Symposium*, November 2010.

[47] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *CGO*, 2011.

[48] T. Sondag and H. Rajan. Static cache coherency analysis. Technical Report 373, Iowa State University, Department of Computer Science, June 2015.

[49] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *Operating Systems Review*, 2007.

[50] G. Upadhyaya and H. Rajan. An automatic actors to threads mapping technique for jvm-based actor frameworks. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 29–41. ACM, 2014.

[51] V.J. Jiménez *et al.* Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09*, 2009.

[52] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT*, pages 307–318, 2010.