Magellan: A Framework for Fast Multi-core Design Space Exploration and Optimization Using Search and Machine Learning

Sukhun Kang and Rakesh Kumar

Coordinated Science Laboratory

1308 West Main St

Urbana, IL 61801

Abstract

As multi-core processor architectures with tens or even hundreds of cores, not all of them necessarily identical, become common, the current processor design methodology, that relies on large-scale simulations, is not going to scale well because of the number of possibilities to be considered. We need intelligent/efficient techniques to navigate through the processor design space.

In this paper, we propose to treat processor design space exploration as a classical search problem. We adapt several well known (and some less known) search/optimization techniques that have been used very successfully in other domains to the problem of efficiently exploring the processor design space. We observe that these techniques result in multi-core processors whose performance is comparable (within 1%) to a processor design that requires an exhaustive exploration of the design space. These techniques often take orders of magnitude *(a factor of 3800 at the minimum) less time for coming up with these processors. We also show that machine learning-based techniques can be applied on top of these search/optimization-based techniques to prune the search space even further.*

We leverage the knowledge gained in this research to develop Magellan – a framework for accelerating multi-core design space exploration and optimization. Magellan can be used to find the highest throughput processors of a given type for a given area, power, or time budget. It can be used to aid even experienced processor designers that prefer to rely on intuition by allowing fast refinements to an input design.

1 Introduction

Multi-core processor architectures are fast entering mainstream due to their scalability, complexity, and energy efficiency benefits. Processors with two, four, and eight cores are already in market. Processors with tens or possibly hundreds of cores may be a reality within the next few years.

As the numbers of cores on a processor increase, not all the cores on these emerging chips may be identical. Heterogeneity in characteristics of cores enables adaptation to diverse workload conditions and characteristics [13, 15]. Future processors may consist of several types of cores on the same die where each core type would be targeted towards specific classes of workloads.

Heterogeneity in core characteristics has a significant implication on design methodology. Conventional methodologies for designing a processor have relied heavily on large-scale simulations to evaluate the various architectural possibilities. However, simulations often take long and can limit the number of possibilities that can be considered for a given time budget. While an exhaustive simulation-based approach has worked for non-multicore processors its limitations get exposed for multi-core architectures, specifically heterogeneous multi-core architectures, due to the increasing (and arguably exploding) size of

[0]University of Illinois at Urbana-Champaign Center for Reliable and High-Performance Computing Technical Report number CRHC-07-05

the design space. For example, designing a 4-core chip multiprocessor where each of the cores can be chosen from a library of 480 cores will consist of evaluating over 2.2 billion possibilities. If evaluating each possibility took a day, it will take close to a million years to find the "best" processor for any reasonable assumption about running simulations in parallel!

This paper recognizes that an exhaustive simulation-based approach to explore the processor design space does not scale well for future heterogeneous multi-core processors. We, therefore, propose to treat the processor design exploration problem as a classical search/optimization problem and adapt some well-known (and some lesser known) algorithms that have been successful in other domains to the problem of finding a good multi-core architecture for a given area/power budget.

While search-based techniques significantly reduce the search space, design space exploration can be accelerated further by using application characteristics to eliminate processors that would be a bad match for the expected workload universe. We propose two machine learning-based algorithms to prune the search space by tagging both the cores that constitute candidate multiprocessors as well as the application with tuples that represent certain application/core characteristics. We observe that such an approach can reduce the search space by up to 13% (9% on average).

We leverage the knowledge gained in this research to develop Magellan – a framework for accelerating multi-core design space exploration and optimization. Magellan is presented with a library of cores that are used to create chip multiprocessors and a set of applications representative of the workload universe. Given this information, Magellan can be used to find the highest throughput processors of a given type for a given area, power, or time budget. It can be used to aid even experienced processor designers that prefer to rely on intuition by allowing fast refinements to an input design.

This paper makes following contributions.

- We show that processor design space exploration can be treated as a classical search/optimization problem and a machine learning problem. We discuss why it adapts well to processor design domain and also our motivation to use our techniques.

- We quantify the numbers of simulation required to explore the design space for various machine learning and search techniques and show that processors that are found using these techniques can have performance within 1% of the highest performing chip multiprocessors found using exhaustive exploration while having up to a factor of 3800 less time overhead.

- We explore the scalability of these techniques for different number of cores and conclude that using intelligent search and machine learning techniques indeed does better than exhaustive search.

- We show that intelligent search and machine learning techniques can be used even for exploring the uniprocessor or homogeneous multi-core architecture design space.

- We present a framework, Magellan, that can be used by industrial architects and academic researchers to accelerate multi-core design space exploration and optimization. Magellan can even aid the experienced designer to help refining an existing design or by validating the existing design.

2 Related Work

This paper proposes to treat design space exploration for multi-core architectures as a classical search/optimization problem and as a machine problem. The methodology presented in this paper would be especially useful for exploring the heterogeneous multi-core architecture design space. There have been several studies [5, 3, 17, 18, 13, 15, 8, 7] studying the power and throughput advantages of heterogeneous architectures. Our study differs from above in that these studies focus on the benefits of an assumed design, and thus give little insight into what constitutes, or how to arrive at, a good heterogeneous design.

The work closest to ours is a study by Kumar *et al*[14] to explore the characteristics of the highest performing chip multiprocessor for a given area and power budget. They demonstrate that the best chip multiprocessors are heterogeneous and non-monotonic (i.e., there is no strict performance ordering among the cores). Their design space exploration is exhaustive and they recommend the usage of efficient techniques to explore the space.

Another related work is by Lee and Brooks[16] who try to minimize the overhead of microarchitectural design space exploration through statistical inference via regression models. Our work differs from theirs in that they use statistical models for predicting and hence comparing performance and power of various architectures while we rely on full-fledged simulations. Hence, there is potentially a speed-accuracy tradeoff that needs to be studied, but is beyond the scope of this work.

Ipek *et al* [11] perform fast multi-core design space exploration by predictive modeling of the the architectural parameters as artifical neural networks (ANNs). This is again a very promising, but somewhat orthogonal approach. Comparing predictive modeling against our approach is the subject of future work.

Strozek and Brooks [26] explore an automated process flow to map a set of applications to a set of heterogeneous core. They also explore application clustering. Their design space exploration is exhaustive, however.

There have been several heuristic-based approaches to search through the design space of a simple uniprocessor or an FPGA soft-core processor. Sheldon *et al* [21] reduce the application-specific core customization problem to a 0-1 knapsack problem and solve it. They also study a synthesis-in-loop approach where design space is explored using impact-ordered trees. Another work by Sheldon *et al* [22] uses DOE (design of experiments) paradigm instead. Padmanabhan *et al* [19] try to cast the design space exploration problem as a binary integer nonlinear programming problem. Pruning techniques are used in [6] to reduce the design space. A combination of analytic performance models and simulation-based performance models is used in [4] to guide design space exploration for sensor nodes. Sherwood *et al* [23] explore the inorder uniprocessor search space using piece-wise linear models and solve their results using integer linear programming. Karkhanis and Smith [12] also explore using an analytical method for doing application-specific superscalar processor design.

There have been several other attempts as well to map a uniprocessor design problem as an optimization problem (too many to cite here). Our work differs from the above in that it is targeted towards the design and architecture of multi-core processors.

While this work tries to minimize the processor design time by reducing the number of instances to be evaluated while still resulting in effective processors, another approach can be making each evaluation of an instance faster during the search. There have been several studies on accelerating simulation. The approaches range from identifying representative portions of program to simulate [24, 9] to using techniques like statistical sampling to approximate full simulation. The reader may want to look up one of the cited papers to get a comprehensive listing of the related work.

3 Treating Processor Design as a Classical Search Problem

This section discusses how the multi-core design space exploration problem can be mapped to a classical search/optimization problem. We discuss the search/opimization techniques that we examined and present the challenges in adapting them to the problem of multi-core processor design space exploration. We also discuss the limitations of these techniques.

3.1 Rationale

Consider the design of a *k-core* chip multiprocessor where each core can be chosen from a library of n cores. There are n^k designs possible. If $n = 100$ and $k = 4$, that is 10 million possibilities. The number of possibilities is 1 trillion and 10 quadrillion for $k = 6$ and $k = 8$ respectively. Each of these possibilities needs to be evaluated for several workloads for al possible mappings of applications to cores for *each* workload. As can be imagined, the design space explodes even for very small values of n and k.

We consider pruning the design space by mapping the processor design space exploration problem to a classical search/optimization problem [20]. We believe that processor design space exploration fits well to a classical search/optimization problem [20] due to the following reasons.

- First, each of the n cores has an area/power cost and a benefit (in terms of performance) for a given application. These cores can be mapped to a two-dimensional space (we treat area and power budget together in this paper) Design space exploration now simply consists of simply *searching* for a combination of *k cores* (representing a $k - core$ processor) that provides the *highest* performance for a given area and power constraint. Performance can be evaluated as an average over *all* workloads that a processor is expected to run. Alternatively, all possible k-core processors can be mapped to a two-dimensional space where the x-axis is the area/power budget and y-axis is the performance of the k-core processor (Section 6 details our performance evaluation methodology). In either case, the 2D space can be treated as a search space for a classical search/optimization problem.

- Second, while there are several k-core processors mapped to the 2D space, there exists only one "optimal" processor (where an "optimal" processor is defined as the processor that has the highest performance for a given set of budgetary constraints). There can be several paths to reach the "optimal" processor from a given starting point processor. Minimizing design space exploration overhead consists of identifying the shortest path from the starting point to the "optimal" processor – this is exactly the goal of classical search/optimization techniques.

- Finally, while the "optimal" processor has the highest performance for a given area/power budget, there can be several other *k-core* combinations that have almost as much performance as the "optimal" while being much easier (or shorter) to reach from the starting point processor. Accelerating design space exploration does not necessarily entail having to reach the "optimal" in shortest possible time – we believe that reaching very close to optimal is often good enough, especially if it takes significantly less time. Not only that, processor design is often dictated by several factors other than performance and budgetary constraints, so an "optimal" processor may not exist. Finding non-optimal, but good solutions has been the focus of several search/optimization techniques.

Because of the above reasons, we considered adapting some classical search/optimization algorithms to the problem of processor design.

3.2 Search Algorithms

This section discusses several search/optimization techniques and provide the pseudo-code and both advantages and disadvantages of each technique.

Exhaustive Search

Exhaustive search for finding the best design involves evaluating all core combinations, account for every permutation of given benchmarks on each combination. This approach ensures that we do indeed find the best combination in each case. While this approach works (barely) for the given workloads and architectural variables, considering more benchmarks and more architectural options will quickly make the exhaustive approach practically impossible. In the following sections, we examine more efficient search algorithms and quantify how closely they come to identifying the best design.

Steepest Ascent Hill Climbing

Hill climbing is a well-known search algorithm that involves, in this case, simply looking for a set of processors better than the currently best processor at each step of exploration. Search is continued until it gets stuck in a local extrema (i.e., we can't find a better processor). Steepest ascent hill climbing involves evaluating all combinations of every neighboring core and using the best k-core combination for the next iteration. A neighboring core is defined as a core that differs in only one parameter and that too in the smallest granularity. For example, two cores that are identical in all parameters except in their icache sizes where the icache size of one is 8KB and the other is 16KB are neighboring cores.

Pseudo-Code

```
1:  S = initial k core configuration E = Evaluation of S MaxIPC = E
2:  BestConf = S
3:  while not stuck do
4:     N = neighbors of S
5:     for  i to maxnum ; maxnum = number of combinations of N do
6:        En = evaluations of N_i
7:        if  En > BestEn; if new N is better than best N so far then
8:           BestEn = En
9:           BestN = N_i
10:       end if
11:    end for
12:    if BestEn > MaxIPC then
13:       MaxIPC = Bestn
14:       BestConf = N_i
```

15: **end if**

16: S = N_i

17: **end while**

The biggest advantage of Steepest Ascent Hill Climbing is its speed. Since SAHC simply tries to go uphill whenever possible without wasting any time evaluating bad or redundant processors, it is relatively fast. However, there is an exponential increase in the number of evaluations as k increases. Another disadvantage of Hill Climbing (including SAHC) is that it may get stuck in local extrema. While trying to maximize performance during current iteration, it might skip some configurations that may lead to solutions closer to the "optimal". Some of these disadvantages can be obviated by using generalized hill climbing that allows backtracking or by applying random start hill climbing, where random starting configurations are used in succession so that multiple paths can be explored.

Genetic Algorithm

Genetic algorithms have been adapted to many optimization problems. Here we try to map them to the problem of multi-core design space exploration. Genetic algorithms involve 4 stages: Reproduce, Crossover, Mutation, and natural selection. Reproduce stage simply evolves the current processor to next neighbors. Crossover stage cross over the population and come up with new processors. Mutation stage randomly picks numbers of cores and mutate them to a random core. Finally, natural selection stage picks 4 top performing processors and use them in the next step of exploration.

Pseudo-Code

1: S = initial k core configuration E = Evaluation of S

2: **for** I = 1 : Kmax **do**

3: RPool = Reproduce (S) ; find better neighbor of each k-core set

4: CPool = CrossOver(ReproducePool) ; crossing over all the populations

5: MPool = 4 Randomly mutated k-core of current population

6: S = Top 4 set of k core configurations out of S, Rpool, Cpool,MPool

7: **end for**

A genetic algorithm fits very well to the problem since the search techniques that GA uses ($crossover$, $reproduce$, and $mutation$) result in efficient and diverse exploration of the solution space without the worry of getting stuck in a local extremum. Specifically, $reproduce$, which always produces the better combination, guarantees that GA will be at least efficient as hill-climbing. Similarly, $crossover$, which tries combining different combinations of current population, and $mutation$, where a fixed number of cores out of k cores are randomly permuted to different cores, ensure that a bigger, more diverse solution space is explored than hill climbing without getting stuck in an extremum.

Our implementation of GA keeps a population of the 4 best k-core processors out of the $reproducepool$, the $crossPool$, and the $mutationPool$, and then uses that population to start the next iteration of search (natural selection). The initial population consists of 4 k-core homogeneous multi-core processor where the 4 core types are as different from the other as possible while satisfying the budgetary constraints Due to the diverse starting configurations, GA quickly finds the range where the "optimal" solution might be and then spends rest of the time evolving toward the "optimal" solution.

Another big advantage of GA over other search techniques is that the same number of k-core processors need to be evaluated for all values of k. So GA has excellent scalability with technology (and number of cores).

One disadvantage of GA (over a quick search algorithm like hill climbing) is that it ends up evaluating several "bad" processors. This is usually acceptable. however, as this enables it to avoid getting stuck in local extrema.

Ant Colony Optimization

Ant Colony Optimization is based on behavior of real ants. Ants move randomly while leaving pheromone behind so the other ants can follow when food is found. Pheromone are volatile and evaporate over time thereby preventing the trailing ants from taking the same path all the time even though food might be gone. This increases the efficiency of search (of food). We adapt this search technique to the processor design space exploration problem taking different paths every iteration in our search for the "optimal" processor (or the ones close to it).

Pseudo-Code

```
 1: S = initial k core configuration E = Evaluation of S
 2: MaxIPC = E
 3: BestConf = S Path = initial path
 4: while not stuck do
 5:    for  i = 1 : Kmax do
 6:       N = better neighbor of path[i]
 7:       En = evaluations of N
 8:       if En > MaxIPC then
 9:          MaxIPC=En
10:          BestConf = N
11:       end if
12:       if En != Eprevious then
13:          Path[i] = [N, En]
14:       end if
15:    end for
16: end while
```

In our implementation of Ant Colony Optimization, the algorithm forces search to take a different path every iteration by randomly moving to a chip multiprocessor consisting of neighboring cores and then continuing the search from there. One advantage of ACO is that it avoids getting stuck at local extrema and explores the solution space more widely compared to hill climbing. A disadvantage is that it take more time because multiple paths need to be explored, each with several iterations over it, to arrive at a good solution.

Hybrid Start Hill Climbing

We also explore a variation of hill-climbing where many starting configurations, including some fixed start configurations and some random ones, are tried. This enables a wider, more diverse exploration of the processor design space.

A disadvantage of HSHC is that it does not scale well when the solution space becomes bigger as more starting configurations are required for the same coverage.

4 Treating Processor Design as a Machine Learning Problem

While the previous section dealt with mapping the design space exploration problem to a search/optimization problem, this section discusses the how machine learning techniques can be used to further prune the search space. We discuss the challenges in adapting machine-learning techniques to the problem of multi-core processor design space exploration. We also present a methodology for coming up with such techniques and the limitations of these techniques.

4.1 Rationale

The processor design space exploration problem can be treated as a machine learning problem due in the following ways:

- First, any "intelligent" search that does not acknowledge the characteristics of the cores in the library as well the characteristics of the benchmarks used to evaluate a core misses an opportunity to prune the search space. To utilize the core and benchmark characteristics to enhance/accelerate the design space exploration, we can tag the cores in our library into different categories (according to various parameters). Similarly, we can tag each benchmark to the appropriate category depending on its characteristic/performance on different cores in the library. Starting configurations in our search can be determined according to the distribution of workloads in each characteristics category. The distribution also determines the combinations of cores to be evaluated. Any processor that does not conform to the distribution of characteristics is not evaluated. Application of intelligent search techniques, such as hill climbing, or genetic algorithm, on top can accelerate the design space exploration even more.
- Second, while trying to figure out the best application to-core mapping for a given combination of cores, a strategy involving trying out all permutations(or mappings) is wasteful considering that some mappings will clearly not be candidates for the highest throughput mapping. With a machine learning based approach, we can categorize our library of cores and the workloads (as discussed above) and try only those mappings (for simulations) where applications and corresponding cores match in their characteristics to the largest extent possible. The best mappings with a given processor and a given set of workloads are likely to be ones where the applications and the cores match in their characteristics.

Thus, machine learning-based approaches reduce both the number of core combinations to be considered as well the number of application-to-core permutations to be considered for a given combination of cores.

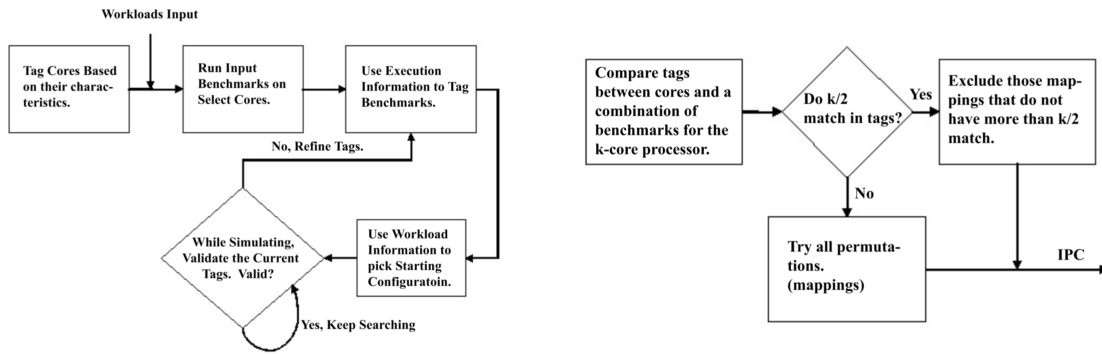

Figure 1. a) Flowchart of Machine Learning Technique And (b) the Optimization of Workloads to Cores Mapping

4.2 Machine Learning-based Search

Machine learning based-search for finding the best processor involves two phases, tagging cores and benchmarks while simulating, and searching. We start out by simulating the benchmarks on a small number of distinct cores that are picked according to their characteristics such that they are sufficient to tag all the benchmarks. The performance and tag values are kept in arrays. As we begin searching through the solution space, we set the starting configuration according to the characteristics of the benchmarks that we have tagged already. Every iteration of search, we look at the combinations of cores that fit the benchmarks' characteristics. As the search proceeds, we run into cores that have not been seen yet. The performance of workloads on these combinations can potentially be used to further refine the benchmark tagging. In this paper, we consider a simpler approach where the tag of the benchmarks do not change after the initial simulations (as discussed above).

To reduce the number of application-to-core mappings considered for a given combination of k cores, we consider simulating only those combination where there are more than $k/2$ matches between characteristics of cores and workloads. Through this exclusion rule, we are able to eliminate a large number of permutations thereby significantly lessening the simulation overhead while still doing a fair evaluation of every core combination.

Note that we can try SAHC or GA on top of above approach which will decrease number of instances considered even more.

Tagging Cores and Benchmarks

We examined two techniques for tagging the cores in the library and the set of benchmarks used to evaluate each core. The first technique is based on core complexity while the second one involves tagging based on core parameters.

Following subsections will examine the methodologies.

1-tuple Tagging: Tagging Based on Complexity

This technique uses the notion of complexity (or complicatedness) of the cores and the benchmarks to tag them (tagging benchmarks is based on the notion of sensitivity to complexity of cores). The tag is a 1-tuple with values - *Simple*, *Moderate*, and *Complex*. For cores, tags are rather intuitive. A core gets the tag *Simple* if it is relatively simple and small in terms of its various parameters. A core is assigned the tag *Complex* when it is relatively complex and big in terms of its parameters. The cores that not clearly *Simple* or *Complex* are marked as *Moderate*.

Benchmarks are tagged according to their performance on our library of cores. A few clearly *Simple*, *Moderate*, and *Complex* cores are picked and each benchmark is run on them. A benchmark that has more or less same performance (defined by a threshold) on a *Simple* core is tagged as *Simple*. If the performance difference is large, the benchmark is tagged *Complex*. If it is neither, the benchmark is tagged as *Moderate*.

The advantage of tagging according to complexity is that we can ignore details regarding the processor parameters and be able to categorize cores and benchmarks into somewhat simplistic categories. For example, it lets us put any type of core or benchmark into three categories which makes the machine learning-based search fairly simple in terms of matching. The limitation is that it lacks the details leading to inefficiency. For example, if a benchmark performs well on all cores that have high fetch width, the machine learning-based search will pick up the cores where not only fetch width is high but other parameters might be complicated as well.

k-tuple Tagging: Tagging Based on Parameters

Another technique that we examined is to tag cores according to their parameters. We categorized cores into 5 categories: Simple, D-cache intensive, I-Cache intensive, Execution units intensive, and Fetch Width intensive. Each tag, therefore, is a 5-tuple where each tuple is one of the fields. Multiple fields can be set at the same time. For example, a core with a large DCache and a large ICache is tagged as (0,1,1,0,0).

Benchmarks are tagged according to their improvements in performance when one of those parameters on a core running the benchmark became complex. For example, *Adpcmc* is execution unit intensive and also fetch width intensive, and is tagged as (0,0,0,1,1).

Note that a more effective tagging technique would be allowing each field in the tag to take parameter values instead of binary values. However, conceptually it is equivalent to the above technique with a larger number of fields per tuple.

The advantage of tagging cores according to parameters is that a lot of attention is paid to the specific parameter(s) that effect(s) the performance of a benchmark. For example, if a benchmark performs well only when the core on which it is being run has a high I-Cache size AND a large number of execution units, the machine learning-based search will be able to pick the right core just for such a benchmark.

A disadvantage of k-tuple tagging is that the search and matching becomes slightly more complex as multiple tags are allowed. In this paper, we declare a match when the characteristics of benchmarks are covered by the core's characteristics.

4.3 Limitations

One limitation of a machine-learning-based approach is that the effectiveness of search and match depends heavily on the goodness and accuracy of tagging. I.e., while a core can be classified as *Simple*, *Moderate*, or *Complex*, for example, it is not clear what is the right threshold for tagging a given benchmark into one of these categories. A rigorous methodology for arriving at a good threshold is needed and is the subject of future work.

5 Magellan Framework

The fact that the problem of multi-core design space exploration can be mapped effectively both to a search problem as well as a machine learning problem entails that alternative approaches can be used for evaluating and optimizing multi-core processors. This section discusses Magellan, a framework for fast multi-core design space exploration and optimization.

5.1 Overall Structure

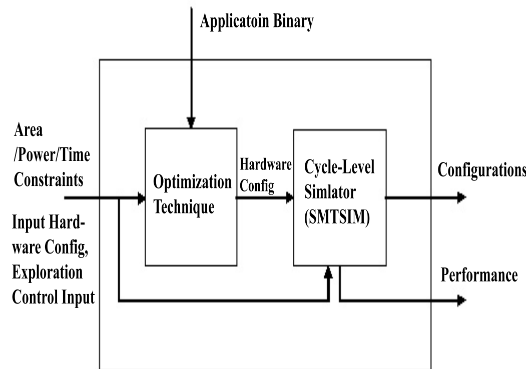

Figure 2. Overall structure of Magellan

The overview figure shows the overall structure of Magellan and the interface between the optimization techniques and the performance simulator. The optimization techniques refer to the search and machine learning techniques discussed in previous sections.

The inputs to Magellan are area, power, time constraints. A time constraint refers to the amount of exploration time that the designer is willing to expend. Magellan also receives as input a set of applications that the designer would want the processor to run effectively on. The output of Magellan is the hardware configuration of a processor optimized according to some objective function and the performance, area, and power characteristics of it.

The extent of hardware diversity can also be provided as a input. So can be an input multi-core that acts as a starting point for search (see (Section 7 for details). Magellan also allows flexibility in terms of constructing multithreaded workloads out of input benchmarks. Evaluations can be done with workloads constructed for different extents of diversity among applications constituting a workload.

Note that while previous discussion has been focused on the design of heterogeneous multi-core architectures, Magellan can be used for the design of al kinds of processors, including uniprocessors and homogeneous multi-processors. (Section 7 shows the effectiveness of search/ML techniques for such processors. Magellan also allows for parameterizations of non-core structures. (Section 5.2 shows how Magellan can be used for exploration when even L2 cache size/configuration is parameterized.

5.2 Usage Models

This section presents possible usages of Magellan. Magellan can either be used as an automated tool multi-core processor design exploration or it can be used as an aid to the experienced designer.

Automated Tool for Fast Multi-core Design

One usage model for Magellan is using it as an automated tool to efficiently explore the processor design space. Some example problem scenarios where we have found Magellan to be useful are:

- Given an area and power budget, find the best k-core processor in the least time,
- Given an area and power budget, come up with the best $k + -\delta$ core processor where δ let's us compare processors with different number of cores,
- Given an area and a power budget, choose the best asymmetric L2 cache configuration for a fully homogeneous processor (or for that matter keep any other core resource constant during different explorations – for example consider only 2-issue cores),
- Given an area and power budget, given a set of hardware optimizations with the associated area and power costs, come up with the best uniprocessor
- Given a time budget, given some area and power budget, find the best $k + -\delta$ core processor as well as the best uniprocessor for different design choices, etc.

For all the above problems, Magellan tries to map the problem to a search/ML problem. Current work involves figuring out other interesting exploration-related problems that Magellan

Aiding the Experienced Designer

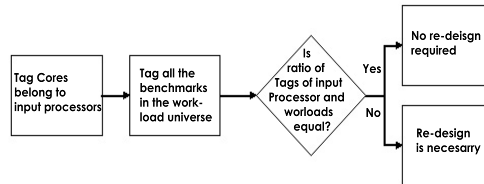

Figure 3. Aiding the experienced designer by complimenting or validating the existing processor

Several experienced processor designers may not rely on an exhaustive search. They would instead rely on their previous designs to come up with an improved processor or search within a smaller library of cores based on their intuition.

In an example scenario that may be common for future many-core heterogeneous processors, a designer may often be sure that at least some cores of certain types should exist on a processor with a large number of cores, but is not sure about what other cores should there be on the processor. For example, even for the throughput-centric (or server) computing market domains, the designer may want at least some complex OOO cores on the processor. In such scenarios, he/she would typically run an exhaustive search with some of the cores fixed while trying to find out what the other cores should be on the processor.

With Magellan, an exhaustive search is not necessary. Our techniques will examine the cores that the designer provided and also the benchmarks which the designer would like to use to evaluate the final processor. Then the cores and the benchmarks are tagged. After tagging is done, an attempt is made to identify the characteristics that the designer-provided cores lack in order to run the benchmarks efficiently. Search involves looking through the library of cores to find core(s) that fit

Issue-width	1 , 2 , 4		
I-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way	L2 Cache	1MB/core, 4-way, 12cycle access
D-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way dual ported	Memory Channel	533MHz, doubly-pumped, RDRAM
FP-IntMul-ALU units.	1-1-2, 2-2-4	ITLB-DTLB	64, 28 entries

Table 1. Various Parameters and their possible values for configuration of the cores.

the characteristic designer-provided cores lacked. These cores are used to create processors that are then evaluated using simulations.

An advantage of this technique is that it reduced the number of simulations in order to come up with a complete processor. A limitation is that the number of simulations do not reduce if the designer-provided cores cover all the benchmark characteristics well (for example, when at least one of the cores on the incomplete processor is complex in all dimensions).

Another way in which Magellan can aid an experienced designer is that it helps validate if a given processor design provides coverage for all the workload characteristics. Validation is done through comparing the ratio between various tags for the cores constituting the given design against the ratio between various tags for the benchmarks that comprise the workload universe. Figure 3 shows the various stages of validation.

A quick validation can be used as a part of the processor design loop as well as processor optimization loop enabling a consideration of significantly more optimizations/features than that are traditionally possible.

5.3 Limitations

One fundamental limitation of Magellan is that it does not provide guaranteed minimum performance, not even probabilistically. While our results (discussed in Section 7) show that our techniques ALWAYS discover processors close to the highest performing processor found using exhaustive search, it is not impossible for a search to go along a wrong path. Note, however, that this problem can be obviated by using the highest performing homogeneous architecture as the starting point for searches. This will ensure a guaranteed minimum performance that is good. The probability of a search going along a wrong path can also be reduced through increasing the number of iterations per search. We are also considering algorithms that provide probabilistic guarantees [2].

Another limitation of the Magellan framework is that it currently takes as input only multiprogrammed workloads, not parallel workloads. However, this limitation is not fundamental and is due to the fact that it is not clear to the authors what is the best strategy for running parallel programs on heterogeneous architectures [5].

Magellan also currently does not account for form factors and aspect ratios of the cores as well as the floorplanning issues, etc. when picking cores for a particular area budget. While we are considering refining our current core packing strategy, it should be noted that Magellan is intended to be used as a first cut to finding good processors. We believe that Magellan is able to do that even in its current form.

6 Methodology

This section discusses the methodology that we used for our evaluations. We discuss details of modeling the various chip multiprocessors, including their area/power requirements and their performance.

Limiting the Baseline Design Space

While treating processor design as a search/ML problem significantly reduces the number of processor instances to be evaluated, each instance still needs to be evaluated for multiple workloads and for all application-to-core mappings for *every* workload. Also, there are still various architectural parameters that can be configured resulting in a large value of n. We make several simplifying assumptions to reduce the number of simulations we had to do for the paper. First, we assume that the performance of individual cores is separable, that is, that the performance of a four-core design, running four applications, is the sum (or the sum divided by a constant factor) of the individual cores running those applications in isolation. This is an accurate assumption if the cores do not share L2 caches or memory channels. Since we are interested in the highest performance that a processor can offer, we assume good static scheduling of threads to cores. Thus, the performance of four particular threads on four particular cores is the performance of the best static mapping. However, this actually represents, in some sense, a lower bound on performance. Prior work has shown that the ability to migrate threads dynamically during execution only increases the benefits of heterogeneity as it exploits intra-thread diversity. To further reduce the number of simulations that we would have to do for this paper, we consider only major blocks to be configurable, and only consider discrete points. For example, we consider 4 cache configurations (per cache) (rather than all the intermediate values). But we consider only a single branch predictor, because the area/performance tradeoffs of different sizes had little effect in our experiments.

Note that none of the above assumptions are fundamental to any of the search/ML techniques discussed in the paper. The techniques discussed in this paper do not assume additivity in terms of the performance of cores of a chip multiprocessor. Similarly, they do not assume zero interaction between cores/caches, or that cores should have private L2 caches, etc. The techniques simply select intelligently the processors that need to be evaluated. The above assumptions are meant to help make the evaluation of a processor be faster.

Modeling of CPU Cores

For all our studies in this paper, we model k-core multiprocessors (for k=4,6, and 8) assumed to be implemented in 0.10 micron, 1.2V technology. Each core on a multiprocessor, either homogeneous or heterogeneous, has a private L2 cache and each L2 bank has a corresponding memory controller. The ITRS roadmap [1] confirms that sufficient pins are available to support four memory controllers for the assumed technology. Assuming private L2 caches reduces the dimensions of the design; evaluating shared caches is the subject of future work.

We consider only in-order cores for this study to keep the design space tractable. Also, the search optimizations, as presented in this paper, might be more applicable to low-end processors as well as to FPGA soft cores that tend to be inorder. We base our processor microarchitecture model on the Alpha EV5 (21164). We evaluate 96 cores as possible building blocks for constructing the multiprocessors. This represents all possible distinct cores that can be constructed by changing the parameters listed in Table 1. The various values that were considered are listed in the table as well. We assumed a gshare branch-predictor with 8k entries for all the cores. The number of distinct 4-core multiprocessors that can be constructed out

of 96 distinct cores is over 3 million, and the number of distinct 6-core multiprocessors that can be made is over 900 million, and the number of distinct 8-core multiprocessors that can be built is over 132 billion.

Other parameters that are kept fixed for all the cores are also listed in Table 1. The various miss penalties and L2 cache access latencies for the simulated cores were determined using CACTI [25]. All evaluations are done for multiprocessors satisfying a given aggregate area and power budget for the k cores. We do not expect the memory and interconnection subsystem to vary significantly with the core type for a given number of cores. We also confirmed that L2's contribution to overall power consumption did not vary significantly between four-core designs taking up the same area, even when the total number of serviced memory requests differed. Hence, we do not concern ourselves with the area and power consumption of anything other than the cores for this study.

In this paper we study only inorder cores, though all the results and analysis applies to out-of-order cores as well. Note that the techniques developed here also apply directly to the problem of mapping soft cores to an FPGA with a fixed budget (LUTs, CLBs, etc.).

Modeling Power and Area

In this paper, the area budget refers to the sum of the area of the k cores of a processor (the L1 cache being part of the core), and the power budget refers to the sum of the worst case power of the cores of a processor. Specifically, we consider peak activity power, as this is a critical constraint in the architecture and design phase of a processor. Static power is not considered explicitly in this paper (though it is typically proportional to area, which we do consider; also comparisons across different budgets are done only for analysis).

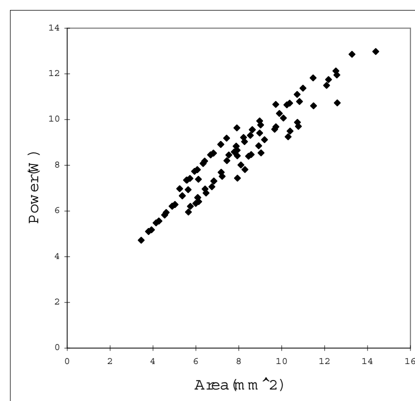

Figure 4. Area and Power of the cores

To model the peak activity power and area consumption of each of the key structures in a processor core using a variety of techniques, we use a methodology identical to [14]. To get total area and power estimates, we assume that the area and power of a core can be approximated as the sum of its major pieces. In reality, we expect that the unaccounted-for overheads will scale our estimates by constant factors (leakage power scaling might not be linear). In that case, all our results will still be valid. Figure 4 shows the area and power of the 96 cores used for this study. As can be seen, the cores represent a significant range in terms of power (4.72-12.98W) as well as area (3.45-14.38mm2).

Program	Description	Suite
ammp	Computational Chemistry	SPEC2000
bzip2	Compression	SPEC2000
crafty	Game Playing:Chess	SPEC2000
eon	Computer Visualization	SPEC2000
mcf	Combinatorial Optimization	SPEC2000
twolf	Place and Route Simulator	SPEC2000
mgrid	Multi-grid Solver: 3D Potential Field	SPEC2000
mesa	3-D Graphics Library	SPC2000
groff	Typesetting package	IBS
gs	PS viewer	IBS
deltablue	Constraint Hierarchy Solver	Olden
adpcmc	Encoder for Adaptive Differential Pulse Code Modulation	MediaBench

Table 2. *Benchmarks used*

Modeling Performance

This section describes the workloads used for evaluation, the performance evaluation methodology.

Workloads

All our evaluations are done for multiprogrammed workloads. Table 2 lists the twelve benchmarks used for constructing workloads. These benchmarks are randomly chosen from the SPEC2000 suite as well as benchmarks from Olden, IBS, and Mediabench suites to ensure diversity. Every multiprocessor is evaluated on two classes of workloads. The *all different* class consists of all possible k-threaded combinations that can be constructed such that each of the k threads running at a time is different. The *all same* consists of all possible k-threaded combinations that can be constructed such that all the k threads running at a time are the same. For example, a,b,c,d is an all different workload while a,a,a,a is an all same workload. This effectively brackets the expected diversity in any workload including server, parallel, and multithreaded workloads. Hence, we expect our results to be generalizable across a wide range of applications.

Evaluation Methodology

As discussed before, there are over 132 billion distinct 8-core multiprocessors that can be constructed using our 96 distinct cores. Similarly, over 900 million multiprocessors for 6 cores. We assume that the performance of a multiprocessor is the sum of the performance of each core of the multiprocessor. Note that this is a reasonable assumption as each core is assumed to have a private L2 cache as well as a memory channel. This is the same architecture (private L2s) assumed in [10] and is supported by recent research comparing private and shared L2 caches for multi-core architectures. We find the single thread performance of each application on each core by simulating for 250 million cycles, after fast-forwarding an appropriate number of instructions [24]. This represents 1152 simulations. Simulations use a modified version of SMTSIM [27]. Scripts are used to calculate the performance of the multiprocessors using these single-thread performance numbers. All results are presented for the best (oracular) static mapping of applications to cores. Note that realistic dynamic mapping can do better [15]. However, evaluating up to over 132 billion multiprocessors becomes intractable if dynamic mapping is assumed.

7 Analysis and Results

In the following sections, we examine the effectiveness of treating processor design as a classical search problem/ a machine learning(ML) problem. We also revisit quantitatively the various usage models for Magellan.

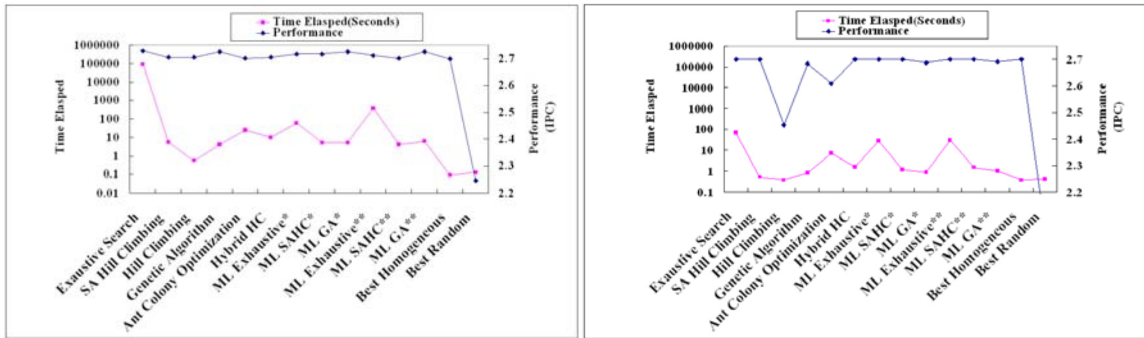

Figure 5. Throughput of the best 4-core multiprocessor discovered by the various techniques, and the time elapsed for each design space exploration. The results are for *all-different* **and** *all same* **workloads for an area budget of 45** mm^2 **and a power budget of 60W**

Figure 5 show the results in finding the best 4-core processor with our techniques compared against the *Exhaustive Search*, the *Best Homogeneous*, and the *Best Random* processor. *Best Homogeneous* is an exhaustive search over only homogeneous multi-core architectures. *Best Homogeneous* for four cores, for example, considers 96 different multi-core architectures (each corresponding to a different core type). *Random Best* corresponds to a randomly chosen multiprocessor that just satisfies the area and power constraint. Note that such a multiprocessor utilizes the available area and power well and hence would perform significantly better than a purely randomly chosen four-core chip multiprocessor.

The results show that the classical search and ML techniques, when adapted to the problem of processor design space exploration, perform exceedingly well. While exhaustive search indeed comes up with the highest performing chip multiprocessor, intelligent search/ML techniques discover processors that are within 0.1 percent of the "optimal" processor in terms of performance. In terms of the time taken for design space exploration, these search/ML techniques have orders of magnitude less overhead. Hill Climbing, for example, is over 168000 times faster than exhaustive search. Similarly, ML Exhaustive is 1600 times faster than exhaustive search. Genetic Algorithm emerges as the best search policy and performs within 0.1% of exhaustive search while being almost 23000 times faster. In fact, all the search/ML techniques are at least 3800 times faster than exhaustive search and perform no worse than 1% in terms of performance! To put these results into perspective, Random Best performs no better than 23% of the exhaustive search in spite of utilizing the area/power budget well.

Therefore, these results demonstrate that using intelligent search/optimization techniques for processor design is a very promising approach.

The benefits of our techniques become much more pronounced as the design space increases. Figure 6 shows the results. As we can see, the overhead of exhaustive search increases exponentially as number of cores increase. The overhead of our techniques, on the other hand, increase only superlinearly in the worst case, and only linearly in the best case. An interesting result also that ML when applied to intelligent search techniques, results in processors with comparable performance, but can often take significantly less time for exploration. For example, ML when applied to Steepest-Ascent Hill Climbing is over 81 times faster than the baseline Steepest-Ascent Hill Climbing.

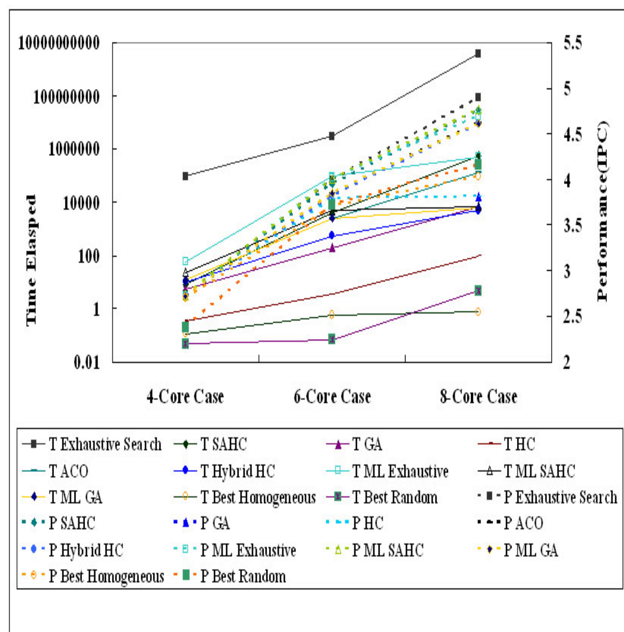

Figure 6. Throughput/time tradeoffs for various techniques for different number of cores for an area budget 50 mm^2 and a power budget of 50W

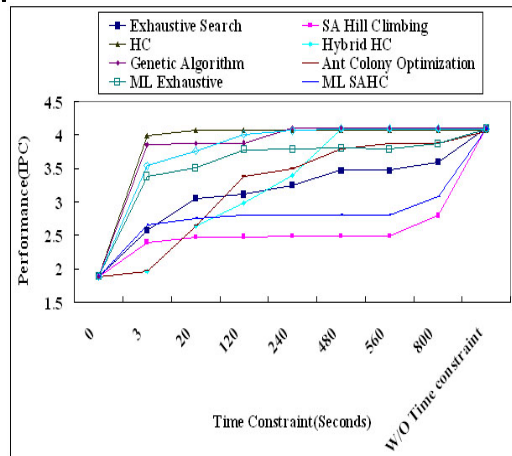

Figure 7. Tradeoffs for 6-core exploration for different fixed time budgets for an area budget of 80 mm^2 and a power budget of 70W

Fast Multi-Core Design/Optimization Using Magellan

In this section we demonstrate quantitatively the effectiveness of Magellan in two usage scenarios.

We studied the effectiveness of using Magellan in terms of finding the highest performing 4-core processor in a fixed time budget. This may represent the case where a designer does not want to expend more than a certain amount of time to processor design space exploration. Figure 7 shows the results for *all different*. As expected, while some techniques evolve slowly towards a better processor (e.g., exhaustive, SAHC, etc.) due to unavoidable evaluation of bad/redundant processors, other techniques (e.g., ACO, HSHC, etc.) evolve faster as they try different starting points. Genetic Algorithm jumps to a high IPC the quickest as one of population might be already in a range of the "optimal" solution.

Another usage scenario that we considered for Magellan is when the designer provides an input CMP and does not want the search algorithms to result in processors that are more than k-hops away. Two processors are considered k-hops away if there exist no more than k cores separating the two most diverse cores on the processor. Processors that are 1-hop away, for example, have cores that are neighbors (defined above) in terms of characteristics. Figure 8 shows the results for *all different*

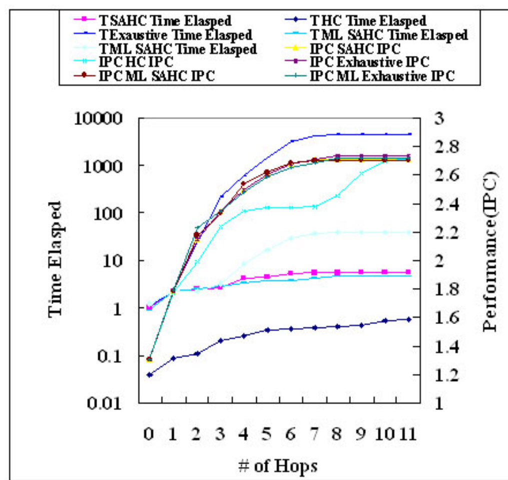

Figure 8. K-step Neighbors Progress Report Starting with an input CMP on budget of an area budget of 45 mm^2 and a power budget of 60W. The designer wants the resulting CMP to be no more than k hops away in terms of characteristics

workloads. As we can see, our search/ML algorithms continue to provide performance equivalent to exhaustive, while taking significantly less time for design space exploration. The results demonstrate that search/ML techniques, when used in this form, can be used to provide fast processor refinements even for an experienced designer who wants to rely on intuition.

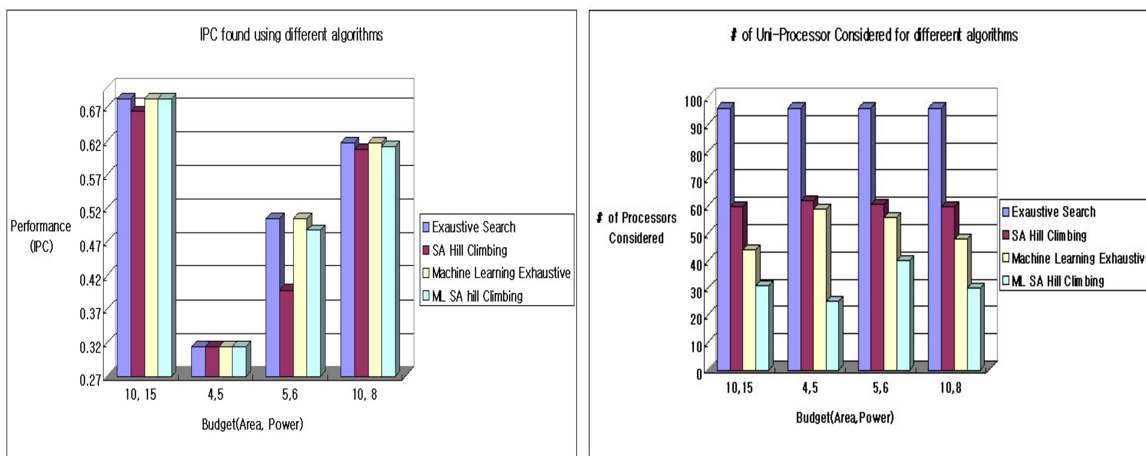

Figure 9. (a) Throughput of the best uni-processor discovered by the various techniques we examined, and (b) the time elapsed for each design space exploration in different budgets.

Generalizing the Magellan Framework

This section applies the search/ML techniques to the problem of uni-processor design for a fixed budget. The algorithms are used to find the highest performing uni-core processor for following budgets: 10 mm^2 and 15W, 4 mm^2 and 5 W, 5 mm^2 and 6 W, and 10 mm^2 and 8W. Exhaustive search tries all possible uni-processors that fit within the budget, while the search/MP approaches consider only a subset of these processors. Figure 9 shows the results. As we can see, the classical search and ML techniques, when adapted to the problem of uniprocessor design space exploration, continue to perform exceedingly well. While exhaustive search indeed comes up with the highest performing uniprocessor, our techniques are able to discover processors that have performance identical to the optimal processors. In terms of the number of processors considered, these techniques reduce the numbers of simulations to almost 30% of an exhaustive approach. Figure 10 shows the results for the case where our techniques are applied for multi-core optimization – core/L2 co-design, specifically. The goal is to again come

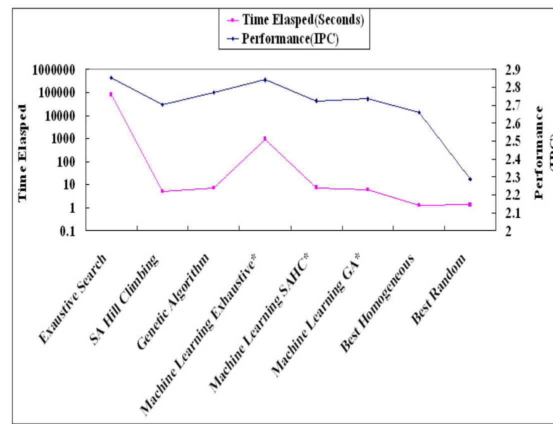

Figure 10. Throughput/time tradeoffs for various techniques for 4-core processor with L2 Cache Parameterized on budget of an area budget of 45 mm^2 and a power budget of 60W.

with the highest performing 4-core chip multiprocessor, except that even L2 cache size is paramterized. I.e., the L2 cache can now be 512KB, 1MB, or 2MB, and depending on the size of the L2, other core resources may be constrained. As the graph shows, the search/ML techniques continue to perform significantly better than an exhaustive search.

8 Summary and Conclusions

This is the first attempt to map the multi-core processor design space exploration problem to a classical search/optimization problem as a machine learning problem. We adapt several well-known (and some less known) search/ML algorithms to look through the processor design space for a given area and power budget. We quantify the corresponding time overhead as well as the performance characteristics of the resulting highest performance processors. We observe that the intelligent search/ML techniques are at least 3800 times faster than exhaustive search and perform no worse than 1% in terms of performance! We observe that these techniques also scale well with the number of cores while the overhead of exhaustive search increases exponentially.

We leverage the knowledge gained in this research to develop Magellan – a framework for accelerating multi-core design space exploration and optimization. Magellan can be used to find the highest throughput processors of a given type for a given area, power, or time budget. It can be used to aid even experienced processor designers that prefer to rely on intuition by allowing fast refinements to an input design. As the number of cores on a processor keep increasing, using intelligent search/optimization techniques may arguably be the only way to be able to do processor design space exploration.

Acknowledgments

The authors would like to thank David Kirk and Ravi Iyer for their useful feedback, as well as the anonymous referees. The research was funded in part by a gift from Intel.

References

[1] International Technology Roadmap for Semiconductors 2003, http://public.itrs.net.
[2] L. Abeni and G. Butazzo. Qos guarantee using probabilistic deadlines, 1999.
[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of International Symposium on Computer Architecture*, 2005.
[4] A. Bakshi, J. Ou, and V. K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, 2005.

[6] D. Fischer, J. Teich, M. Thies, and R.Weper. Efficient architecture/ compiler co-exploration for asips. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[7] S. Ghiasi and D. Grunwald. Aide de camp: Asymmetric dual core design for power and energy reduction. In *University of Colorado Technical Report CU-CS-964-03*, 2003.

[8] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *Proceedings of IEEE International Conference on Computer Design*, 2004.

[9] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. In *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[10] J. Huh, S. W. Keckler, and D. Burger. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[11] E. Ipek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGARCH Comput. Archit. News*, 34(5):195–206, 2006.

[12] T. S. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In *In International Symposium on Computer Architecture, June 2007*, 2007.

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

[14] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[15] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture*, June 2004.

[16] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS*, 2006.

[17] T. Morad, U. Weiser, and A. Kolodny. ACCMP - assymetric cluster chip-multiprocessing. In *CCIT Technical Report 488*, 2004.

[18] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. In *Computer Architecture Letters, Vol 4*, 2005.

[19] S. Padmanabhan, R. K. Cytron, R. D. Chamberlain, and J. W. Lockwood. Automatic application-specific microarchitecture reconfiguration. In *13th Reconfigurable Architectures Workshop (RAW)*, Rhodes Island, Greece, Apr. 2006.

[20] E. Rich and K. Knight. *Artificial Intelligence, 2nd Edition*. Morgan Kaufmann, 1991.

[21] D. Sheldon, R. Kumar, F. Vahid, R. Lysecky, and D. Tullsen. Application-specific customization of parameterized fpga soft-core processors. In *International Conference on Computer-Aided Design, ICCAD*, 2007.

[22] D. Sheldon, F. Vahid, and S. Lonardi. Soft-core processor customization using the design of experiments paradigm. In *International Conference on Design and Test in Europe, DATE*, 2007.

[23] T. Sherwood, M. Oskin, and B. Calder. Balancing design options with sherpa, 2004.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2002)*, Oct. 2002.

[25] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *Technical Report 2001/2, Compaq Computer Corporation*, Aug. 2001.

[26] L. Strozek and D. Brooks. Efficient architectures through application clustering and architectural heterogeneity. In *CASES*, 2006.

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