A Study of Dynamic Phase Adaptation Using a Dynamic Multicore Processor

A Study of Dynamic Phase Adaptation Using a Dynamic Multicore Processor

PAUL-JULES MICOLET, University of Edinburgh
AARON SMITH, University of Edinburgh and Microsoft Research
CHRISTOPHE DUBACH, University of Edinburgh

Heterogeneous processors such as ARM's big.LITTLE have become popular for embedded systems. They offer a choice between running workloads on a high performance core or a low-energy core leading to increased energy efficiency. However, the core configurations are fixed at design time which offers a limited amount of adaptation. Dynamic Multicore Processors (DMPs) bridge the gap between homogeneous and fully reconfigurable systems. Cores can fuse dynamically to adapt the computational resources to the needs of different workloads. There exists multiple examples of DMPs in the literature, yet the focus has mainly been on static partitioning.

This paper conducts the first thorough study of the potential for dynamic reconfiguration of DMPs at runtime. We study how performance varies with static partitioning and what software optimizations are required to achieve high performance. We show that energy consumption is reduced considerably when adapting the number of cores to program phases, and introduce a simple online model which predicts the optimal number of cores to use to minimize energy consumption while maintaining high performance. Using the San Diego Vision Benchmark Suite as a use case, the dynamic scheme leads to $\sim 40\%$ energy savings on average without decreasing performance.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; *Heterogeneous (hybrid) systems*;

ACM Reference format:
Paul-Jules Micolet, Aaron Smith, and Christophe Dubach. 2017. A Study of Dynamic Phase Adaptation Using a Dynamic Multicore Processor. 1, 1, Article 1 (September 2017), 19 pages.
DOI: none

1 INTRODUCTION

Chip Multicore Processors (CMP) are now ubiquitous in embedded computing as single threaded performance improvements have slowed. CMPs have to be carefully designed, balancing the size of each core with the total number of cores on the chip. Larger cores are typically good at exploiting instruction level parallelism (ILP) but might potentially be very power hungry. Smaller cores on the other hand require less power but offer limited performance, forcing software developers to parallelize their code with multiple threads, which is a tedious process. As the size and the number of cores is fixed at design time, choosing the right balance is difficult [5, 23].

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
This work has been supported by Microsoft Research through its PhD Scholarship Programme.

DOI: none

Asymmetric Chip Multicore Processors (ACMP) have been proposed [14] to overcome this issue. These processors feature either different sized cores [10] or different Instruction Set Architectures [24] to efficiently tackle a multitude of different workloads. Dynamic Multicore Processors (DMP) push this further by introducing Core Fusion [9]. Similar to ACMPs, Core Fusion allows the chip to have different sized cores, but this can be changed at runtime. In a DMP, cores can be fused dynamically to create larger cores similar to a superscalar processor. Any number of cores can potentially be combined together whenever a workload exhibits a large amount of ILP. When a program exhibits low ILP, the DMP can decouple fused cores to conserve energy.

While a large number of DMPs have been proposed in the literature [9, 12, 16, 26], these efforts focus on the hardware and microarchitectural design. They evaluate the hardware using a fixed number of fused cores or provide an oracle for dynamic fusion. There exists little [13] to no literature on predicting core fusion from a software perspective. To the best of our knowledge, there has been no study on dynamically changing the number of cores fused to better match the phases of a workload in a homogeneous DMP compared to ahead of time fusion.

We start with an explanation of the theoretical limitations of core fusion and what we can expect in terms of performance. We then discuss how classical loop optimizations such as unrolling can have a large impact on performance when fusing cores. Using the San Diego Vision Benchmark Suite [25] (SD-VBS) as a use case, we show that programs exhibit various phases with different amounts of ILP. We then perform a limit study on the potential for decreasing energy consumption while maintaining performance when adapting the number of cores for each program phase. Our results show that using dynamic core fusion can save up to 42% on average while maintaining the same level of performance as a fixed number of cores. We also show how latency introduced by reconfiguring the system can influence the impact of core fusion. Finally, we build a simple online model using linear regression that predicts the optimal number of cores per phase for reducing energy consumption while maintaining performance. This practical model leads to an average of 37% saving in energy with no performance loss.

To summarize, our contributions are:

- We analyze the limits of core fusion using an analytical model.
- We study the loop optimizations required to ensure efficient use of core fusion.
- We offer an in-depth comparison of static and dynamic core fusion schemes on the San Diego Vision Benchmark Suite.
- We show that core fusion has the potential to offer a large reduction in energy savings.
- We show how a simple linear-regression based model can predict the number of cores to fuse for different program phases.

2 DYNAMIC MULTICORE PROCESSOR

Dynamic Multicore Processor. DMPs contain hardware which can be modified post fabrication. Mitall's survey [14] defines three types of modifiable resources: the core count [9], number of resources that each core has [8] and microarchitectural features [1, 7, 21]. In our paper we focus on DMPs that modify the core count.

EDGE ISA. We assume a DMP similar to TFlex [12] using an Explicit Data Graph Execution [3] (EDGE) instruction set architecture (ISA). EDGE ISAs encode dependencies between instructions at the ISA level. Code is organised as blocks of instructions where all instruction communication is local to the block [20]. Each block has a single entry point but may have multiple exits. This enables the architecture to dispatch, speculatively, blocks with low overhead [12, 18], therefore, increasing exploitation of ILP.

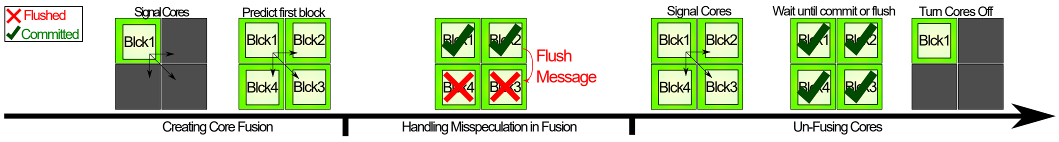

Fig. 1. Core Fusion Mechanisms for our EDGE-based architecture.

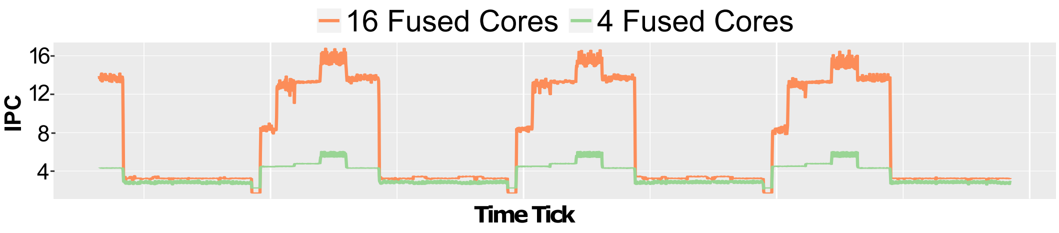

Fig. 2. IPC of a typical benchmark (Disparity from SD-VBS) when executing on a fused 4 or 16 cores processor.

Core Fusion. Core Fusion is achieved by fusing a set of *physical* cores to create larger *logical* cores. This does not modify the physical structure of the chip, instead it provides a unified view of the group of physical cores to the software. For example, fusing two cores generates a logical core with twice the amount of execution units, register files and L1 cache. Fusion is a dynamic modification and may occur during the execution of a program to better fit the workload. Unlike traditional CMPs, fused cores will operate on the same thread and attempt to extract Instruction Level Parallelism (ILP) rather than Thread Level Parallelism (TLP) [13, 16]. Figure 1 shows the different stages and mechanisms of core fusion for a four core system. When creating a new core fusion a master core informs all other cores about the fusion and submits predicted block addresses to the cores. When a core mispredicts a branch in a fusion, it informs the other cores which flush any younger blocks. When un-fusing, the master core informs all other cores. Once the cores in the fusion have committed or flushed their blocks they are turned off and the master core continues to fetch blocks from the thread. The extra hardware required to support dynamic reconfiguration is very minimal [12] since most of the machinery already in place can be reused such as the cache coherence protocol when fusing and un-fusing the cores. We discuss this in further detail in Section 5.

3 MOTIVATION

This section motivates the use of dynamic core fusion and its impact on performance and energy. It also shows that loop optimizations have a significant performance impact when fusing cores.

3.1 Dynamic Core Fusion

When discussing core fusion, previous work has focused on delivering speedup results using core fusion [9, 12] and [13] demonstrated how to predict static core fusion. A static fusion will fuse cores into a single logical core (LC) and execute a thread on this new core. As evident from that work, fusion improves the performance of the program by maximizing speed. However, as we will show, static core fusion may not be the perfect match for all situations.

Figure 2 plots the Instruction Per Cycle (IPC) performance variation over the execution of the *Disparity* Benchmark [25] on a fused 4-cores and 16-cores processor. On 4 cores, the performance oscillates between an IPC of 2 and 6 depending on the phase while on 16 cores the IPC can be as high as 16. More importantly, for some phases (half of the time), the same level of IPC is achieved (~3) whether the program runs on 4 or 16 cores. A DMP could exploit this by fusing only 4 cores during

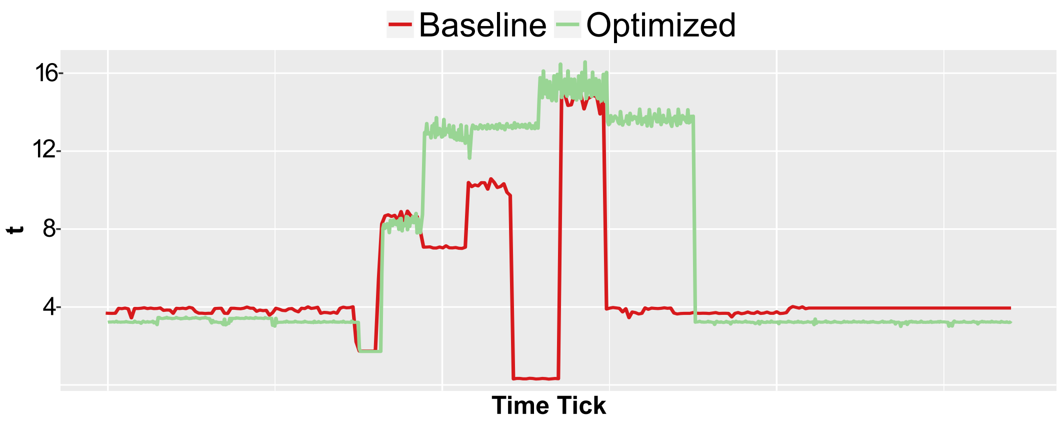

Fig. 3. Impact of loop transformations on fused cores for the Disparity benchmark.

these low IPC phases and fusing 16 cores during the high IPC phases to maximize performance while minimizing energy consumption.

3.2 Code Optimizations

When cores are fused they execute blocks of instructions in parallel on each physical core in the LC. In order to obtain the best results, we must generate large blocks as this leads to a higher IPC on the LC [13]. The optimizations we apply include aggressive loop unrolling, inlining and replacing conditional statements with either software predication or architecture-level predication. These optimizations are well known and do not require any structural modifications of the program.

Figure 3 illustrates the impact of applying loop transformations compared to a standard compiler not specifically tuned for an EDGE architecture. As can be seen, the impact of these transformations can be large in some cases and are absolutely necessary to sustain a high IPC for a long enough period. More details about the loop transformations are given in section 6 but this example illustrates the need for careful tuning of the compiler to achieve high performance on such architecture.

3.3 Summary

This section has shown that programs exhibit phases with various amount of ILP available. A dynamic multicore processor can take advantage of this property to fuse a large number of cores for the high-ILP phases and fuse a smaller number of cores when ILP drops in order to save energy. We have also illustrated the importance of fine-tuned code transformations in order to achieve sustained performance and increase the potential for fusing cores. The next section will study in more details the expected impact of core composition using an analytical model.

4 A STUDY OF CORE COMPOSITION

In this section we study the performance limitations of fusing several cores into a single logical core (LC). This allows us to better understand what leads to good performance and how to determine regions of code that benefit from core fusion. The two major obstacles to gaining performance with core fusion are branch prediction and synchronization costs.

4.1 Branch Prediction

As seen in section 2, DMPs use fused cores by speculatively executing blocks of instructions on the fused cores [16, 18]. This puts a strain on the branch predictor since efficiently using the fused

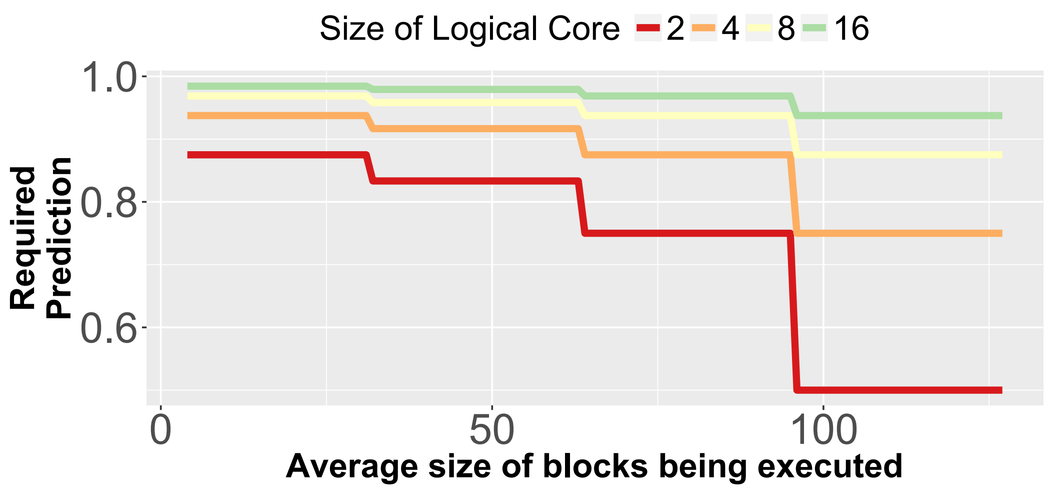

Fig. 4. Required Prediction Accuracy for a Logical Core Size to be efficient given an average size of basic blocks.

cores depends on the miss-prediction rate. The branch predictor has to meet a different accuracy requirement depending on the size of both the LC and average size of a block being executed. Given a Logical Core *LC* of size *i*, denoted LC_i, we can determine the minimum branch prediction requirement using Formula 1:

$$min_{PredLC_i} = \frac{(BlocksInFlight \times i) - 1}{BlocksInFlight \times i} \tag{1}$$

where *BlocksInFlight* represents the number of total blocks being executed on *LC*. *BlocksInFlight* will vary depending on the average size of the blocks, the largest size of a block a lane can carry (*MaxBlockSize*) and number of lanes each physical core has.

$$BlocksInFlight = NumOfLanes - \left\lfloor \frac{AverageBlockSize}{MaxBlockSize} \right\rfloor \tag{2}$$

When a program is running on an LC, one of the blocks will always be unconditionally executed, which is why we require one less block to be predicted.

Figure 4 shows the expected prediction accuracy required to ensure the full utilization an LC given an average size of the blocks in flight. We can see that adding extra physical cores to an LC requires an increasingly accurate branch predictor, especially when the size of a block is under 50 instructions. This informs us in two ways; first of all large LCs will need to run on code sections with less control flow as they are more sensitive to branch misspredictions. Second of all, branch prediction can be a simple method of evaluating the current effectiveness of an LC. Given a certain number of cores, if the prediction accuracy is under the limits presented in Figure 4 we can easily determine that the LC is sub-optimal.

4.2 Synchronization Cost

In order for a program to execute correctly, the cores in a logical core (LC) must communicate when they have finished executing a block [18]. This ensures that the cores have fetched blocks from the correct branch path and that the data predictions are correct. This requires that blocks are committed in a sequential fashion with the non-speculative block committing first and the most

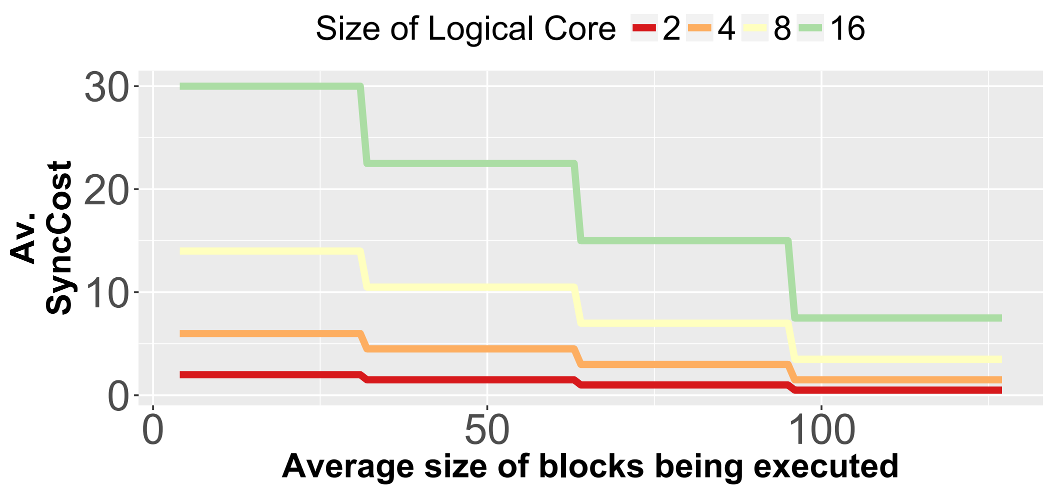

Fig. 5. Synchronization Cost in cycles for a given number of cores in a composition and an average block size.

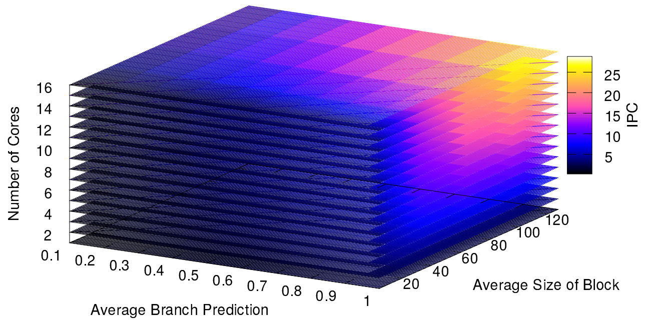

Fig. 6. IPC Estimate given a core composition size, average branch prediction and average block size for a dual-issue core system. A higher IPC means better performance.

recent speculative block committing last. If all the lanes of a core are full then it must commit a block before it fetches a new one. A core may have to wait for all other cores to have committed before fetching a new block, we define the worst-case estimate of this stall as the **Synchronization Cost**. The Synchronization Cost, in cycles, is defined in equation 3 and is measured by averaging the overall number of cycles each core in a composition will be waiting until it can continue to fetch and execute new blocks. As previously mentioned, this is a worst-case estimate as block sizes will fluctuate during the execution of a program.

$$SyncCost_i = \frac{\sum_{n=0}^{i-1} \left(Lanes - \left\lfloor \frac{AverageBlockSize}{MaxBlockSize} \right\rfloor \right) \times n}{i} \tag{3}$$

Figure 5 shows how many cycles the Synchronization Cost will be for a given LC and average block size. As we can see, the larger the block the lower the Synchronization Cost is. This is due to the fact that cores fetch less blocks, thus they wait less for other cores in the composition to finish committing theirs. We can also see that large compositions running small blocks have a high Synchronization Cost. This clearly indicates that larger LCs must be avoided when dealing with smaller blocks as the Synchronization Cost outweighs code execution.

4.3 Summary

Using the information we have gathered we determined the worst-case estimation of IPC for a composition using Average Block Size, Average Branch Prediction, and Synchonization Cost. We represent this information in Figure 6 where we can see the worst-case estimate IPC performance of a composition when we assume that each core can execute a block at an IPC of 2. From what we previously learned, Figure 6 shows us that in order to obtain optimal performance we will need a high branch prediction accuracy and large blocks. It also shows us that larger compositions can easily under-perform; for example we can see that 16 core-compositions often have IPCs under 15, meaning that each core has an IPC under 1.

5 EXPERIMENTAL SETUP

The previous section studied the performance potential for core fusion using an analytical model. We now present the experimental setup used for the remaining parts of the paper where we conduct a thorough evaluation of core fusion with a cycle-level simulator.

5.1 Benchmarks

For this paper we study the performance of a Dynamic Multicore Processor (DMP) on a set of vision benchmarks designed for hardware and compiler research [25]. The San Diego Vision Benchmark suite (SD-VBS) is composed of nine single-threaded C benchmarks ranging from image analysis to motion tracking. These benchmarks represent state-of-the-art applications in image and vision recognition that are prevalent in embedded systems.

Vision applications have typically very regular and simple control flow which enables the formation of large blocks of instructions with a single-entry point. Our processor relies on the ability to form large blocks to exploit ILP which makes these applications particularly well suited. As the results will show, the phase length does not have much of an impact on energy savings when the reconfiguration overhead is low.

5.2 Architecture and Simulator

We use a cycle-level simulator of an EDGE-based Dynamic Multicore Processor [19]. The simulator is configured to model a 16 core multiprocessor, with 32 KB private L1 caches and a shared DRAM. Each core has a 128 entry instruction window and blocks are mapped in 32 instruction chunks to the window. This allow several combinations of block sizes in the window between one, 128 instruction block, to four, 32 instruction blocks. Each core is configured to issue two instructions per cycle from any of the blocks in the instruction window. When all cores are fused there may be a maximum of 64 blocks in flight (16 cores x 4 blocks per core).

The architecture and core fusion mechanics are similar to the work described in [12, 18]. To ensure the accuracy of the simulator it is validated against an RTL implementation of the processor. This validation is done by running workloads on RTL either in simulation or an FPGA and comparing the traces cycle by cycle with the software simulator. The simulator uses Ruby from Gem5 [2] as its memory subsystem. Parts of the fusion mechanism have been modeled in RTL, allowing us to also validate it.

5.3 Fusing Cores

In this processor, the micro-architecture is distributed: register files, Load Store Queues (LSQs), L1 caches and ALUs all look like nodes on a network. This means that when we fuse cores together, this is similar to adding an extra node to our network. When cores are fused, one of the cores will be executing a non-speculative block from a single thread whilst all other cores execute speculative blocks that are predicted from the same thread. We use a simple round robin policy to choose which core is going to execute a speculative block. Whenever the blocks are committed we also pick a new core to execute the next non-speculative block. When we start a new thread on a fused core the OS and runtime write the new core mapping to a system register. The hardware then flushes these cores if they are not idle and sets the PC of the first block of that thread on one core in the core fusion and starts executing.

Fusing cores is therefore a very lightweight process. We estimate that switching the size of the logical-core (LC) results in a delay of 100 cycles on average. The actual time varies based on the time it takes the cache coherence protocol to move the data around the memory system. Section 8.5 discusses in more details how latency affects energy efficiency and shows that dynamic core fusion is still highly beneficial even when considering overheads of 1,000 cycles.

5.4 Compiler

Each benchmark is compiled with Microsoft's Visual C++ compiler for EDGE, with O2 optimisations and instruction predication for hyperblock formation [20].

5.5 Measuring Performance and Power

We run 5 simulations per benchmark, one for each LC sizes 1, 2, 4, 8 and 16. For each composition we record the IPC of the LC at an interval of 640 committed blocks. We selected 640 committed blocks as it allows each core in an LC to execute enough blocks before taking the measurement. This is due to the fact that the highest LC of 16 cores can execute up to 64 blocks at a time, thus recording performance after 640 blocks allows each core to have executed at least 10 blocks. Using committed blocks as an interval allows us to easily compare each simulation as the total number of committed blocks does not change even if the compositions are different.

Due to the fact that we use the EDGE ISA [20], we cannot use McPAT to model power consumption as it differs from traditional CISC/RISC cores modeled in McPAT. Instead we use a coarse grained power model, either a core is turned on or or it is off.

6 CODE OPTIMIZATIONS

This section describes optimizations focused on reducing control flow and expanding block sizes which is necessary for high performance as seen in section 4.

6.1 Loop Unrolling

Loop unrolling is a very common optimization used to reduce the overhead of the loop header and to better expose Instruction Level Parallelism (ILP). When dealing with very tightly-knit loops, logical cores may perform poorly due to the fact that they execute many small blocks, thus increasing the Synchronization Cost. Unrolling loops will both reduce the number of blocks required to execute the loop and increase the size of the blocks, thus reducing the Synchronization Cost and increasing ILP. For example, the innermost loop in Figure 7 should be completely unrolled and its outer loop unrolled partially to increase the block size. There are certain factors which can limit the usefulness of loop unrolling.Based on our current architecture, we may not have more than 32 load or store instructions per block. Therefore, if we unroll memory intensive loops, we must ensure we do not go above this threshold. Going above this threshold leads to creating a new block which will put a strain on the Instruction Cache. Another issue is that unrolling loops with conditional statements may not help improve the size of the block as the conditional branches might still segment the new blocks. So we should avoid unrolling such loops.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
     for(int k = 0;k < 5; k++)
         a[i][j] = a[i][j] * b[k][j];
```

Fig. 7. Example of an inner-most loop which should be completely unrolled.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
      a[i][j] = a[i][j-1]
                          * b[i][j];
```

Fig. 8. Example of a data dependency which can be removed by interchanging the loops.

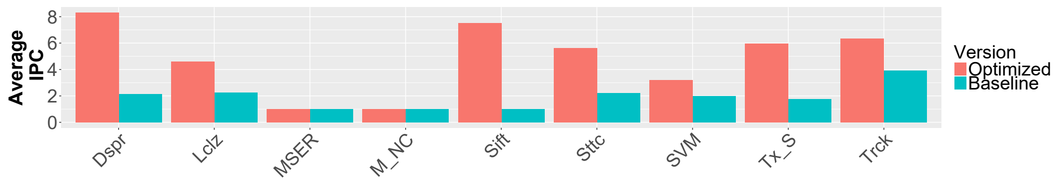

Fig. 9. Average IPC using the optimal sized logical-core, with and without optimizations. Higher is better.

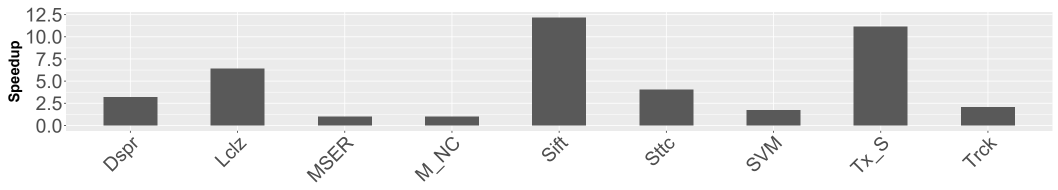

Fig. 10. Speedup from using code-optimizations over baseline source code using the same optimal sized logical-core.

6.2 Loop Interchange

When dealing with nested loops there is one reason we have determined for interchanging the loops. The case arises when interchanging the loop removes dependencies in the inner-most loop. The dependency in Figure 8 can be removed by interchanging the loops. This allows us to unroll the inner loop efficiently, but also remove any kind of dependency between blocks; Since two blocks from the same loop may execute on different cores, we want to reduce any kind of data dependency, minimizing core communication.

6.3 Predication and Hyperblock Formation

The EDGE architecture must split blocks whenever control-flow is present [20]. If a loop contains a conditional statement the loop block has to be split into two if there is no predication. Hyperblocks aim to reduce branching by fusing two or more blocks into a single predicated block [20]. Hyperblocks both reduces synchronization cost and branch prediction requirement as discussed previously. This is especially important in control-flow intensive loops where unrolling increases the number of conditional statements.

6.4 Results

While the optimizations described above and their tuning would be easy to implement in the compiler, we did not, unfortunately, have access to the compiler's source code. We therefore manually modified the source code of our benchmarks by manually interchanging or unrolling loops. In the case of predication and hyperblock formation, we simply converted simple if-then-else statement into ternary operators whenever possible. We also tried to reorder the statement within the body of the loop to avoid having control flow in the middle of the body. We then verified that

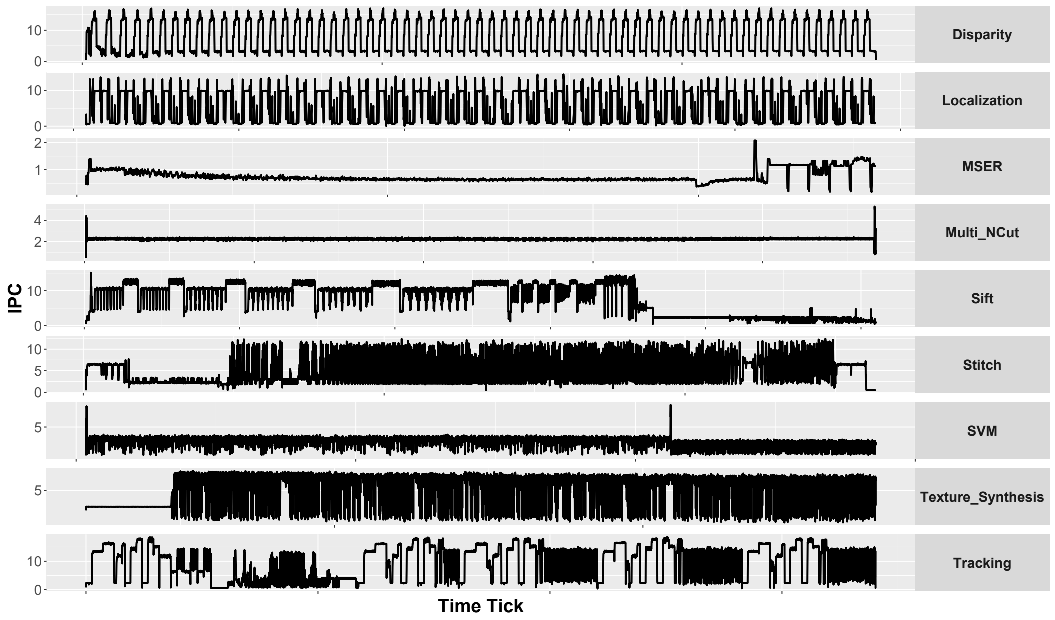

Fig. 11. IPC as a function of time for each benchmark when run on 16 fused cores.

our source code modification had the intended effect by dissembling the binary produced by the compiler. We modified between 0 and 12 loops depending on the benchmark.

We compare the best static core fusion using the optimized code with the unmodified code, both version compiled with -O2. Figure 9 shows the resulting IPC for the baseline case and the optimized benchmarks when run on a core with the optimal number of fused core to maximize performance. The IPC of the baseline is very low for the majority of the benchmarks which might give the impression that core fusion is rather inefficient. However, after applying the simple optimizations described above, the average IPC is significantly increased in many cases.

Since optimizations change the total number of instructions, we also show the actual speedup obtained using cycle count in Figure 10. As we can see, benchmarks *MSER* and *Multi-NCut* do not perform any differently. This is due to the fact that none of these optimizations can be successfully applied on these benchmarks. For the other benchmarks we see significant improvements of up to 12× for *Sift* when the optimizations are applied.

6.5 Summary

Overall, this section shows that classical loop transformations can have a large impact on the performance of fused cores. Without these optimizations, it would be more difficult to motivate the use of core fusion even at a static level as the average IPC does not deviate enough from a single core.

7 BENCHMARK EXPLORATION

This section explores how core fusion affects the performance of the SD-VBS benchmarks. We first perform a phase analysis, followed by a study of the IPC variation for static core fusion. We then motivate the use of dynamic core fusion using the information gathered.

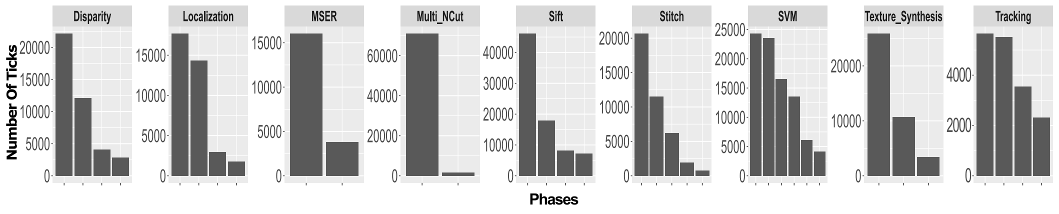

Fig. 12. Number of phases determined for each benchmark using kMeans clustering and their distribution.

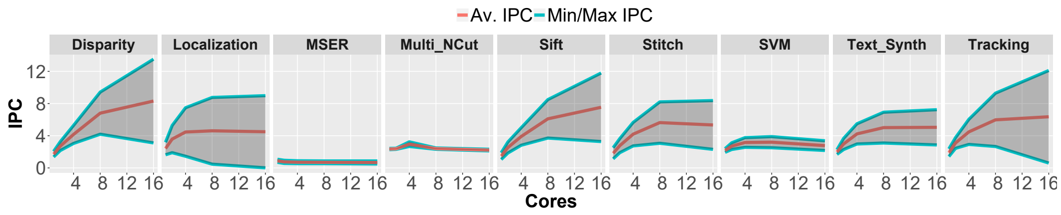

Fig. 13. Comparing average, smallest and greatest IPC for each SD-VBS benchmark using logical-core size of 16.

7.1 Phase Detection

Figure 11 present the IPC performance through time for all the benchmarks when using a logical core (LC) composed of 16 cores. The IPC is calculated for each time tick, which is set at interval of 640 blocks committed. The IPC varies a lot for some of the benchmarks such as *Disparity* or *Localization* where we expect dynamic fusion to be especially good. For other, such as *Multi_NCut*, the execution is dominated by a single long phase with constant IPC, which will clearly show no benefit from using dynamic fusion.

To better understand how dynamic core fusion improves performance, either by improving speedup or reducing energy, we first study how each benchmark features different phases during their execution. For every benchmark we regroup the IPC results of 16,8,4,2,1 fused cores and use kMeans clustering to determine phases. This process is only done for the purpose of exploring this set of benchmarks. Intervals that exhibit similar IPC values when run on different core counts are classified in the same cluster. In order to find the correct number of clusters we plot the Sum of Square Errors (SSE) for a given cluster size from 1 to 15 and determine the optimal cluster to be in the elbow in the plot [6].

Figure 12 shows us the optimal number of clusters for each benchmark and the frequency of each cluster. The data can be corroborated with the information found in Figure 11. For example, benchmarks *MSER* and *Multi_NCut* feature two phases, with one dominating phase. This means that it will be impossible to obtain any kind of performance improvements through dynamic reconfiguration. For all the other benchmarks, they each have at least two dominant phases. Since each phase is a cluster of similar IPC values, having two or more clusters will result in a higher chance of benefiting from dynamic core fusion.

7.2 Static Core Fusion Exploration

Figure 13 shows how the average Instructions Per Cycle (IPC) changes as we increase the size of an LC, going in powers of 2 from 1 to 16 fused cores. We see that, for most benchmarks, fusing more cores provides an increase in IPC performance. Benchmarks *Disparity*, *Localization*, *Sift*, *Stitch*, *Texture Synthesis* and *Tracking* all at least observe a speedup of 2x when using core fusion.

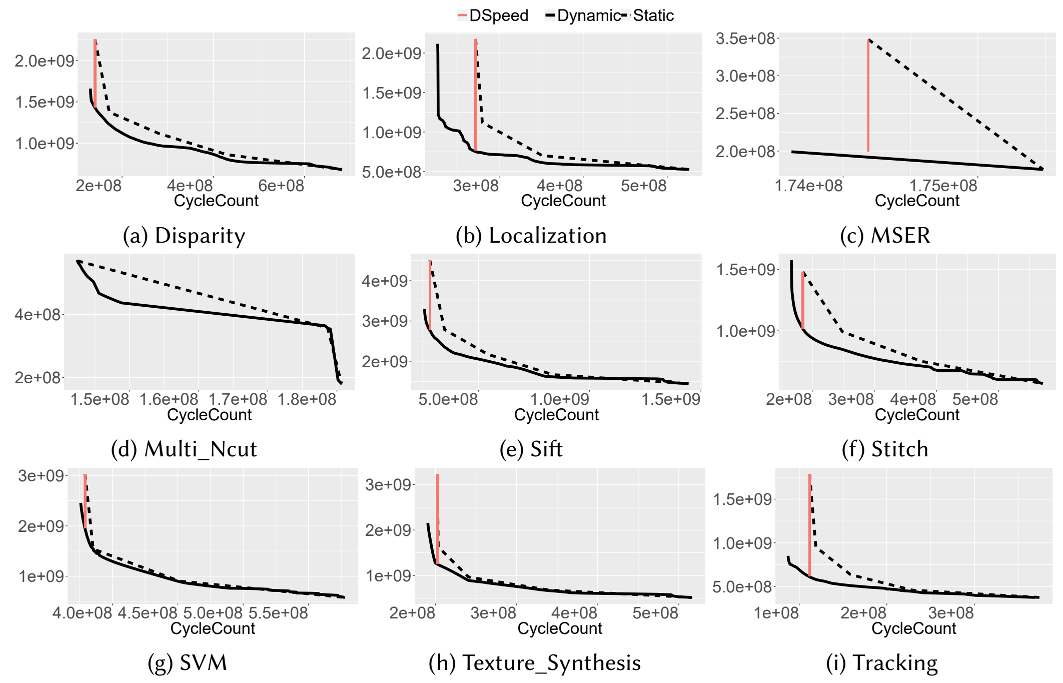

Fig. 14. Time (x-axis) vs. Energy (y-axis) tradeoffs using Static and Dynamic Composition Schemes.

However increasing the size of an LC is not always beneficial as can be seen in benchmarks *Localization*, *MSER*, *Multi_NCut*, *Stich*, and *SVM*. For benchmarks *Localization* and *Stitch* the performance increases when fusing up to 8 cores, where-as *MSER* and *Multi_NCut* never benefit from core fusion. Referring back to Figures 11 and 12, we can see that *MSER* and *Multi_NCut* feature one dominating long phase, both performing poorly. This explains the lack of scaling for these two benchmarks.

Figure 13 also shows the standard deviation of the IPC for each given LC size represented by the grayed out areas. For example, running the *Disparity* benchmark on a LC of 16 cores, we have an average IPC of 8.3 with a standard deviation of 5.2. The standard deviation for 16 cores shows that the performance can drop down to 2.5. An IPC of 2.5 when using 16 cores is very inefficient as this represents 0.1 of an instruction per cycle for each core. We can observe that when using a LC of size 4 for the *Disparity* benchmark we achieve an average of 4.1 with a standard deviation of 1.2. Thus, if the logical-core could change size, there is a possibility that we could reduce the overall energy consumption of the system by switching from 16 to 4.

Overall, most benchmarks that benefit from large logical-cores will also be met with important standard deviations of IPC performance. The high standard deviation is evidence of performance phases found in each application which are likely to benefit from dynamic adaptation.

8 DYNAMIC CORE FUSION

Having studied the behavior of our program under a fix number of cores, we now study the impact of varying the number of fused cores throughout program execution. We first describe how we generate traces for the dynamic core fusion schemes. Before we begin the analysis we define two types of static core fusion:

- **Static Benchmark**: A fixed fused-core which is optimal for the benchmark at hand (SB).

- **Static Suite**: A fixed fused-core which represents the average best for the entire suite of benchmarks. This represents our baseline for the paper (SS).

We then compare the static fused-core scheme with the results obtained for the dynamic one for the SD-VBS benchmarks. This is followed by a closer analysis for two dynamic core fusion objectives: one that optimizes speed and another that optimizes for efficiency.

8.1 Creating Dynamic Core Fusion Traces

With dynamic core fusion, we have the ability to change the number of cores for each time tick (an interval of 640 blocks) during program execution. In order to explore the different performance and energy trade-off that is possible to achieve with this technique, we collect traces of execution for the whole application. We run the whole application on 1,2,4,8 and 16 fused cores and record for each time tick the cycle count. Using these 5 traces, we can then reconstruct any arbitrary dynamic execution and generate dynamic traces.

To simplify the exploration process, time ticks of the same phase will always be attributed the same number of cores. This is done to reduce the search space as on average we have 48494 ticks which would result in an average of $5^{48,494}$ different possible executions.Since the maximum number of clusters found is 6 (for SVM), we only build a maximum of $5^6 = 15625$ different dynamic execution traces. When we switch the size of the logical core (LC), we use the performance of that LC from its respective trace file and add an extra 100 cycle penalty for switching the size of an LC. With all these different dynamic core fusion traces, we can now find the optimal schemes for maximizing speed or maximizing efficiency.

8.2 Dynamic Core Fusion

Figure 14 shows the trade off between time and energy using either a static scheme fixed once at the beginning of the program or a dynamic scheme. The dotted line represents the static core fusion scheme whilst the solid line represents the Pareto Front of all the dynamic core fusion traces. The vertical line represents the amount of energy that can be saved from using a dynamic core fusion scheme that matches the same speed as the best static scheme.

Figure 14 demonstrates how static core fusion fails to maintain good energy efficiency as we improve speed. For example, *Disparity* (Figure 14a) is fastest on 16 fused cores, but has an 1.63x increase in energy consumption for a 1.22x improvement in speed. When using the dynamic scheme, it is clear that energy consumption increases at a slower rate when increasing speed. In this case the number of cores is adapted to the current program phase, using just enough cores to maintain high performance without wasting energy.

8.3 Optimizing for Speed

In this section we define our dynamic scheme to be one that matches the same speed performance as the fastest static core fusion for the benchmark: **DSpeed**. This is equivalent to the vertical line found in Figure 14. This scheme enables us to maintain good performance whilst reducing energy consumption drastically.

Figure 15 shows the speedup of **DSpeed** and SB and the respective energy consumption. The results are normalized against the performance of SS, which is 8 cores fused. The SS core count is obtained by averaging the number of cores for each benchmark using the SB scheme. The speed performances are the same for SB and **DSpeed** as the dynamic scheme is designed to match the static speed. We can see that some benchmark perform better when using benchmark specific core fusions rather than SS. Both *Disparity* and *Sift* obtain a 1.25x speedup when using the SB scheme whilst *Tracking* benefits from a 1.10x speedup.

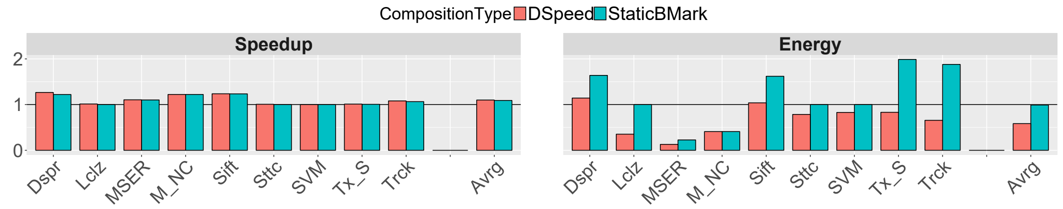

Fig. 15. Maximizing speed for all the SD-VBS benchmarks. For Speedup, higher means better, for Energy, lower is better.

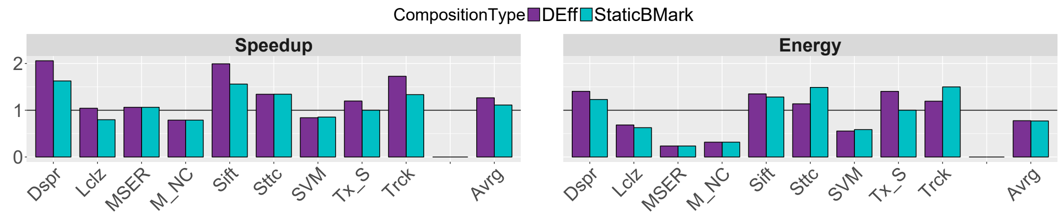

Fig. 16. Maximizing efficiency for all the SD-VBS benchmarks.

When looking at the Energy graph of Figure 15, we can clearly notice where the SS scheme fails. Benchmarks *MSER* and *Multi_NCut* feature very little improvements when using core fusion, therefore the SS will perform very poorly when it comes to energy consumption for the benchmarks. SB does not always perform well neither; as we can see, for the benchmarks *Disparity*, *Sift*, *Texture_Synthesis* the energy consumption is much higher. This is due to the fact that these benchmarks perform best on a 16-core system, however as we saw in Figure 13, the variation in performance always increases when fusing this many cores. The **DSpeed** scheme always performs better than the SB scheme and can even match the SS scheme on energy consumption whilst improving speed such as in the *Sift* benchmark. For the *Localization* benchmark, the **DSpeed** matches the performance of both the SB and SS whilst reducing energy consumption by 65%.

Overall, by using **DSpeed**, we can reduce energy consumption by 42% compared to both SB and SS without impacting performance. This illustrates the greatest advantage of using a DMP since the number of fused core can be adapted continuously depending on the amount of ILP available for each phase.

8.4 Optimizing for Efficiency

In this section we define our dynamic scheme to maximize the efficiency metric EDD, which is defined as $Energy \times Delay \times Delay$ where Delay is the execution time. This metric attempts to optimize speed whilst remaining energy efficient; we call the scheme **DEff**. Figure 16 shows the speedup performance of **DEff** and SB and their respective energy consumption. The results are normalized against SS which is a fixed-composition of 4 fused cores.

Unlike the previous results in Figure 15, we can see that there are differences in the speedup obtained by **DEff** and SB. For benchmarks *Disparity*, *Sift* and *Tracking* the **DEff** scheme is 1.30x faster than the SB scheme and at least 1.75x faster than the SS scheme. It is important to note that this extra speedup does not incur great increases in energy consumption compared to SB: only 1.10x for *Disparity* and *Sift*. In fact, for *Tracking* **DEff** saves 20% in energy compared to SB. When comparing to SS **DEff** is 1.75x times faster for only 19% more energy for the *Tracking* benchmark.

Overall, **DEff** results in a 1.25x speed increase compared to SB and SS whilst consuming 25% less energy than SS. This shows how dynamic core fusion's flexibility allows us to get better speedups whilst not drastically increasing our energy consumption.

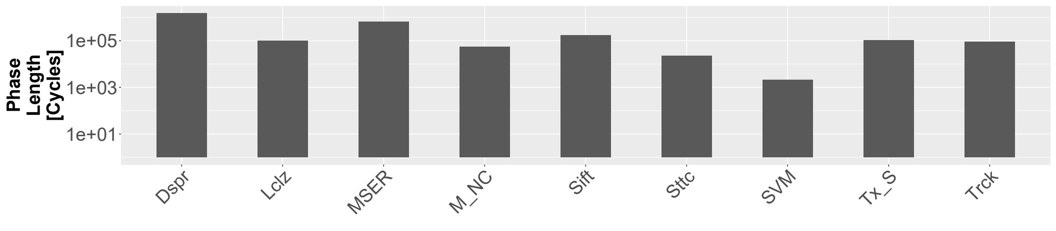

Fig. 17. Average number of cycles without switching.

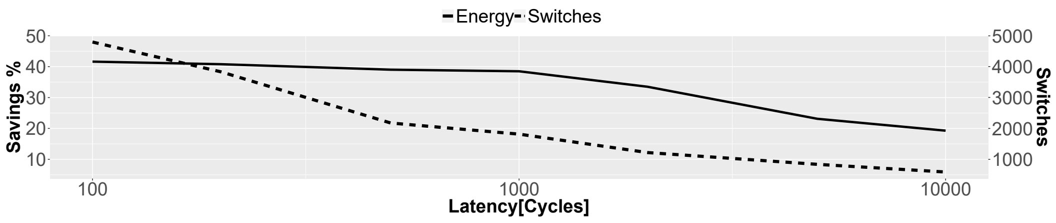

Fig. 18. Energy savings and number of switches as a function of the switching latency in cycles.

8.5 Reconfiguration Latency

Up until now, the paper has assumed a reconfiguration latency of 100 cycles whenever dynamic reconfiguration occurs as explained in section 5.3. This section studies the impact of a larger reconfiguration overhead on energy savings. First, figure 17 shows the average phase length for each benchmarks when maximizing energy savings while maintaining performance (**DSpeed**). As can be seen, the majority of the benchmarks run for long period of several ten of thousands cycles before any switching occurs. Therefore, we expect that even if the reconfiguration latency would be increased to larger value (*e.g.*, 1,000 cycles), its impact might be minimal.

Furthermore, we always have the option to reconfigure less often, in the case where a change in configuration only brings marginal reduction in energy. In such case it might be more beneficial to keep running on the slightly less optimal configuration than paying a cost for reconfiguration. Figure 18 illustrates perfectly this scenario, showing how energy behaves as a function of the reconfiguration overhead (averaged across benchmarks). For each latency value, we determine the best trace of reconfiguration to keep performance constant while minimizing energy (**DSpeed**). The left y-axis expresses the energy savings relative to the static scheme, while the right y-axis shows the total number of switches. The energy savings remains high up to a latency of 1,000 cycles, with a noticeable decrease in the number of switches. For latency values over 1,000 cycles, the energy savings drop considerably, with very few switching occurring. This data shows that even if the reconfiguration overhead is 1,000 cycles, average energy savings of 38% are possible compared to 42% when the overhead is 100 cycles.

8.6 Summary

Overall, we have seen that whether we optimize for speed or efficiency, dynamic core fusion will always lead to higher speedup or lower energy consumption than a fixed configuration. This is due to the presence of phases in applications that the dynamic scheme can exploit to reduce wasting energy in low ILP phases. We have shown that maximizing speed can be highly energy inefficient when using a static LC and that a dynamic scheme can help reduce energy consumption by 42% on average. When optimizing for efficiency, we have shown that a dynamic scheme can help improve

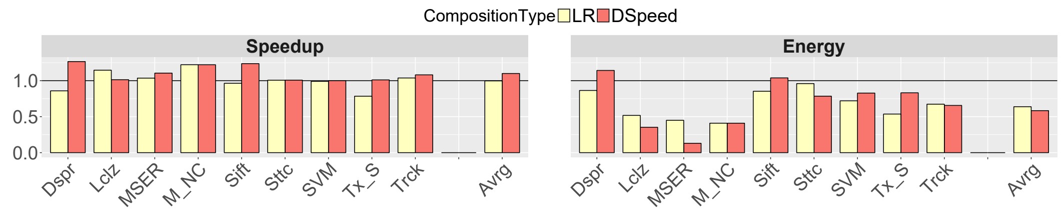

Fig. 19. Performance results for maximizing speed for the SD-VBS benchmarks using our linear regressor (LR) model.

both speed and energy consumption, for example in the case of *Tracking*, and overall we can improve speed by 1.25x whilst saving 25% energy.

9 LINEAR REGRESSION MODEL

Having shown the potential that a DMP has to offer, we now present a simple scheme that is used to exploit the large energy savings available. The main idea is to monitor at runtime some performance counters and make a decision at a regular interval on how to reconfigure the cores. For this purpose, we train a model offline using the data collected and presented earlier in the paper. Once trained, the model predicts the optimal number of cores based on the performance counters from the previous time interval and reconfiguration occurs if it is different from the current number of cores.

9.1 Model

We use a linear regressor which makes predictions using a simple weighted sum of the input features, which is very lightweight and easy to integrate in hardware. The model is trained offline using the traces gathered from our prior analysis for the **DSpeed** scenario which maximizes energy savings while maintaining performance. The dataset consists of a set of four input features (average block size, and percentage of integer, floating point and load operations) and the optimal number of cores for each time tick for each program. These features were chosen as they are easy to extract from the hardware. To speedup the learning process, we create a single data point per phase, averaging the features of all the ticks in a phase, resulting in a total of 34 pairs of optimal core number and features.

The training consists of finding the weights that minimize the error when predicting the optimal number of cores to use across all time ticks and benchmarks. Since we have only considered core configurations which use a power of two number of cores, the linear model is built to predict the logarithm (base 2) of the number of cores. The prediction is rounded up to the nearest integer in the interval $[0, 4]$. The following equation represents the trained linear model which can be used to make prediction:

$$log_2(\textbf{\#cores}) = -7.7 \;+\; 0.028 \cdot \textbf{avgBlkSze} \;+\; 0.075 \cdot \textbf{\%int\_ops} \;+ \\ 0.069 \cdot \textbf{\%fp\_ops} \;+\; 0.21 \cdot \textbf{\%ld\_ops}$$

For instance, if we observe at runtime an average block size of 6 instructions, and 77%, 1% and 18% of integer, floating point and load operations, respectively, then the predicted value will be 2.092. Rounded up to the nearest integer value, 2, the optimal number of cores predicted will, therefore, be 4. As can be seen, the largest weight is on the percentage of loads operations. This is due to the fact that loads can be fired independently to the Load-Store Queue. Unlike stores that depend on previous memory instructions blocks being committed, loads can be fired with less overhead. As data can be speculatively fetched, loads instructions can receive data from other cores before the data is stored, speeding up the load instruction. By increasing the core count on load heavy blocks this will improve performance more reliably due to cores being able to issue loads in parallel.

9.2 Results

To evaluate the performance of our model, we use leave-one-out cross-validation, a standard machine-learning methodology which tests the model using not seen during training. For instance, if we want to test the model for one program, let say *Disparity*, we train the model using the dataset from all the other programs combined. We then use the resulting trained linear model to predict the optimal core number for each time tick of the disparity program and report the performance achieved.

Figure 19 shows the performance in terms of speed and energy that is achieved using our linear model normalized by a fixed static configuration. The fixed configuration maximized performance across all the benchmarks using 8 cores and is the same as in the previous results presented in figure 15. On average, our linear regressor model is able to consume 37% less energy compared to the 8 cores fixed configuration and is able to exactly match its speed.

The performance is also compared with the best possible choice of dynamic reconfiguration, **DSpeed** which acts as an Oracle. As can be seen, the linear model is able to exploit similar energy savings to the **Dspeed** scheme in most cases. On average it reduces energy by 37%, which is within 5% of the 42% achievable by the **Dspeed** scheme. These results show that it is possible to implement a simple realistic lightweight scheme which offers large energy savings.

10 RELATED WORK

Reconfigurable Processor. ElasticCore [21] proposes a morphable core. They present a core that uses both dynamic voltage/frequency scalign (DVFS) and microarchitectural modifications such as instruction bandwidth and capacity. They propose a linear regressor model to determine reconfiguration, however it uses more runtime information than ours, such as branch prediction and cache misses. Overall Tavana et al's architecture is 30% more energy efficient than a big.LITTLE architecture. In [4] they also propose a similar core architecture that modifies microarchitectural features. They provide extensive analysis of SPEC 2000 benchmarks and demonstrate that with machine learning and dynamic adaptation they can double the energy/performance efficiency compared to a static configuration. MorphCore [11] focuses on reconfiguring a core for thread level parallelism. It switches between out-of-order (OoO) when running single threaded applications and an in-order core optimised for simultaneous multi threading (SMT) workloads. This provides an opposite solution to our DMP: providing a large core made for ILP that can be modified to better fit TLP workloads. MorphCore outperform a 2-Way SMT OoO core by 10% whilst being 22% more efficient.

All these projects focus on uni-core modifications, and traditional CISC/RISC like architecture which differs from our work.

Dynamic Multicore Processors. Previous work on Dynamic Multicore Processors includes Bahurupi [16, 17], and CoreFusion [9]. These architectures use a standard ISA and either fetched fixed sized instruction windows [9] or an entire basic blocks [16]. Other DMPs such as TFlex [12] and the E2 DMP [18] use the custom ISA called EDGE [20].

Dynamic Core Fusion. In the work of Pricopi et al. [17], they show how dynamic reconfiguration is beneficial when it comes to scheduling tasks. However, they do not discuss any method of automatically deciding the optimal core-composition beyond a 4 core fusion. Instead they use speedup functions determined from profile executions of applications to determine how to schedule tasks. They also do not discuss what software characteristics help determine when to reconfigure the cores, or how to optimise software. Work on using machine learning to automatically choose a composition was achieved in [13]. However this work does not involve changing the core fusion during the execution of the benchmark. Also, the machine learning model focuses on using high-level information from StreamIt's [22] language constructs.

Voltage Scaling. Voltage scaling is another method of reducing energy consumption [15], however this approach is orthogonal to DMPs. Whilst both methods adapt to programs phases, DMPs can also be used to speed up the execution of programs.

11 CONCLUSION

In this paper we have shown that whilst static core fusion already demonstrates promising results, it becomes harder to be efficient when increasing the size of logical cores. We explained theoretical limitations of static core fusion; without high branch prediction and large blocks, it under-performs. This was followed by a study of a suite of benchmarks, showing how performance varies greatly on a static logical core.

We then created two dynamic schemes: **DSpeed** that matches the speed of the fastest static core fusion and **DEff** that maximizes efficiency. Using these schemes we saw that **DSpeed** saves on average 42% energy compared to the optimal static logical core for a given benchmark. We also showed that **DEff** can improve speed performance by up to 1.30x and reduce energy consumption by 1.20x on some benchmarks. Finally, we developed a simple linear regressor model that can be used to decide on the number of core to fuse at runtime in order to optimize for performance, leading to a 37% drop in energy while maintaining the same level of performance as a static scheme.

REFERENCES

[1] L. Bauer, M. Shafique, S. Kreutz, and J. Henkel. 2008. Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set. In *the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, 6.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[3] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, X. Chen, R. Desikan, S. Drolia, J. Gibson, M. S. Govindan, P. Gratz, H. Hanson, C. Kim, S. K. Kushwaha, H. Liu, R. Nagarajan, N. Ranganathan, R. Reeber, K. Sankaralingam, S. Sethumadhavan, P. Sivakumar, and A. Smith. 2004. Scaling to the End of Silicon with EDGE Architectures. *Computer* 37, 7 (July 2004), 44–55.

[4] C. Dubach, T. M. Jones, and E. V. Bonilla. 2013. Dynamic Microarchitectural Adaptation using Machine Learning. *ACM Transactions on Architecture and Code Optimization* 10 (2013). Issue 4.

[5] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. 2012. Exploring and Predicting the Effects of Microarchitectural Parameters and Compiler Optimizations on Performance and Energy. *ACM Transactions on Embedded Computing Systems* 11S, 1, Article 24 (June 2012).

[6] Leese Everitt, Landau. 2001. *Cluster Analysis.*

[7] C. Fallin, C. Wilkerson, and O. Mutlu. 2014. The Heterogeneous Block Architecture. In *IEEE 32nd International Conference on Computer Design (ICCD)*. 386–393.

[8] H. Homayoun, V. Kontorinis, A. Shayan, T. Lin, and D. M. Tullsen. 2012. Dynamically Heterogeneous Cores Through 3D Resource Pooling. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, 1–12.

[9] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. 2007. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, 186–197.

[10] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley. 2016. Portable Performance on Asymmetric Multicore Processors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '16)*. ACM, 24–35.

[11] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. 2012. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[12] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, Divya Gulati, D. Burger, and S. W. Keckler. 2007. Composable Lightweight Processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*

(MICRO 40). IEEE Computer Society, 381–394.

[13] P. J. Micolet, A. Smith, and C. Dubach. 2016. A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors. In *Proceedings of the 17th ACM Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES '16)*. ACM, 113–122.

[14] S. Mittal. 2016. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *Comput. Surveys* 48, 3, Article 45 (Feb. 2016).

[15] S. Pagani, A. Pathania, M. Shafique, J. J. Chen, and J. Henkel. 2017. Energy Efficiency for Clustered Heterogeneous Multicores. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (May 2017), 1315–1330.

[16] M. Pricopi and T. Mitra. 2012. Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture. *ACM Transactions on Architecture and Code Optimization* 8, 4, Article 22 (Jan. 2012).

[17] M. Pricopi and T. Mitra. 2014. Task Scheduling on Adaptive Multi-Core. *IEEE Transactions on Computer* 63, 10 (Oct. 2014), 2590–2603.

[18] A. Putnam, A. Smith, and D. Burger. 2011. Dynamic Vectorization in the E2 Dynamic Multicore Architecture. *SIGARCH Comput. Archit. News* 38, 4 (Jan. 2011), 27–32.

[19] A. Smith and A. Bakhoda. 2017. Microsoft Research Development Kit for EDGE Architectures. https://www.microsoft.com/en-us/research/project/e2/. (2017). Accessed: 2017-07-14.

[20] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. 2006. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*.

[21] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun. 2015. ElasticCore: Enabling Dynamic Heterogeneity with Joint Core and Voltage/Frequency Scaling. In *ACM/EDAC/IEEE Design Automation Conference (DAC '15)*. 1–6.

[22] W. Thies and S. Amarasinghe. 2010. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, 365–376.

[23] E. Tomusk, C. Dubach, and M. O'Boyle. 2015. Four Metrics to Evaluate Heterogeneous Multicores. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article 37 (Nov. 2015).

[24] A. Venkat and D. M. Tullsen. 2014. Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. 121–132.

[25] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, 55–64.

[26] Y. Watanabe, J. D. Davis, and D. A. Wood. 2010. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, 2–13.