Towards Automatic Program Partitioning

Sean Rul
Ghent University
Sint-Pietersnieuwstraat 41
B-9000 gent, Belgium
srul@elis.ugent.be

Hans Vandierendonck
Ghent University
Sint-Pietersnieuwstraat 41
B-9000 gent, Belgium
hvdieren@elis.ugent.be

Koen De Bosschere
Ghent University
Sint-Pietersnieuwstraat 41
B-9000 gent, Belgium
kdbosche@elis.ugent.be

ABSTRACT

There is a trend towards using accelerators to increase performance and energy efficiency of general-purpose processors. Adoption of accelerators, however, depends on the availability of tools to facilitate programming these devices.

In this paper, we present techniques for automatically partitioning programs for execution on accelerators. We call the off-loaded code regions *sub-algorithms*, which are parts of the program that are loosely connected to the remainder of the program. We present three heuristics for automatically identifying sub-algorithms based on control flow and data flow properties.

Analysis of SPECint and MiBench benchmarks shows that on average 12 sub-algorithms are identified (up to 54), covering the full execution time for 27 out of 30 benchmarks. We show that these sub-algorithms are suitable for off-loading them to accelerators by manually implementing sub-algorithms for 2 SPECint benchmarks on the Cell processor.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Design, Performance

Keywords

partitioning, sub-algorithms, accelerators, off-loading

1. INTRODUCTION

Currently, much focus is placed on *accelerators* as a useful way to increase computational strength and efficiency of existing processors. These accelerators can be integrated on-die, as in STI's Cell processor [20] and the POD accelerator [28], or they may be realized in accelerator boards as in GPUs [18], ClearSpeed's CS301 [12] and Nallatech's Slipstream FPGA-based accelerator [4].

The problem with accelerators, however, is programming them. Indeed, programmers must partition their programs in a portion that is executed on the main processor and a portion that is off-loaded to the accelerator. Accelerating applications on specialized cores requires several steps, ranging from high-level program analysis up to low-level optimization specific to the accelerator core (Figure 1). The amount of work to perform in each step depends strongly on the accelerator's architecture, the programming models used and the strength of the compiler. For instance, most accelerator architectures have private memories [12, 18, 20], implying that the used data structures must be copied-in or -out, or they must reside solely in the accelerator's private memory. The size of the task is also important, due to communication delays and other task startup overheads. Finally, accelerators are typically strong on data-parallel applications, so (i) the presence of data parallelism in the accelerator's task is a plus and (ii) the code must be restructured to exploit the data parallelism. This code migration path is very time-consuming and error-prone [22], so tool support is essential for making the use of accelerators widespread.

Some of the steps dealing with low-level representations of the program have already been partially automated. E.g. implementing the program partitioning can be as simple as inserting pragmas in the CellSS [3] and Cellgen [19, 22] programming models and compilers can aid in restructuring code and data for execution on accelerators [6]. Identifying a good program partitioning, however, requires extensive program analysis, especially if control flow is complex and if a large number of data structures or global variables is used. For program partitioning, however, tool support exists at best for debugging particular partitionings [15] and for timing validation [13]. The goal of this work is to facilitate the task of program partitioning, by suggesting good partitionings to the programmer and by automatically identifying which data structures must be copied-in or -out, or can remain local to the accelerator's private memory.

The program partitioning problem is, in general, non-trivial. Current successes reported for accelerator cores typically apply to applications with regular data flow and little control flow, e.g. string matching on the Cell processor [27], LDPC on a GPU [10] and multiple sequence alignment [26]. In these applications, program partitioning is fairly simple and can be performed manually. It suffices to isolate the most time-consuming loops in a program and to accelerate these, as in the FLAT and CIGAR approaches [15, 25].

CF'09, May 18–20, 2009, Ischia, Italy.

As accelerator cores become more prevalent in the future, however, it will become essential to apply program partitioning also to less regular applications, e.g. featuring many function calls, complex control flow and less regular data flow. For these applications, simple heuristics, such as identifying hot loops, are not sufficient at all, as control flow may frequently enter and leave the hot loops. Approaches based on min-cut network flow [8] are also not sufficient, as they suffer from lack of scalability.

In this paper, we introduce a framework that enables automatic identification of good program partitionings of control-intensive applications. Hereto, we introduce the notion of *sub-algorithms*, parts of the program that can easily be separated from the rest of the program. The contributions of this paper are:

1. We present a *theoretical framework* to reason about program partitioning for control-intensive applications in Section 2. This theory builds on inter-procedural control flow graphs and data flow graphs, as the program partitioning must consider both control flow and data flow to minimize communication.

2. We propose three heuristics to track data flow in Section 2.2. First, the private use of data structures as it makes communication of the private data structures unnecessary. Second, we consider the amount and size of data structures that are shared with code executing on the main processor. Third, we consider the data traffic to decide on suitable sub-algorithms.

3. Evaluation of the proposed heuristics (Section 3) shows that on average up to 12 sub-algorithms suitable for program partitioning are found in a mix of SPECint2000 and MiBench benchmarks. This shows that even small benchmarks contain significant opportunities for program acceleration. Furthermore, we apply the techniques to partition the bzip2 and mcf benchmarks in Section 4. Implementing a partitioning of them on the Cell processor shows the validity of the approach.

Besides program partitioning, we see other potential applications of sub-algorithms in the areas of benchmarking and program comprehension. These applications are motivated in Section 5. In Section 6 the conclusions of this work are summarized and potential extensions are discussed.

2. METHODOLOGY

The topic of this paper is to complete the work flow for program partitioning (Figure 1) by automatically providing potential program partitionings to the programmer. The additional steps in the work flow required by our approach are shown dashed in Figure 1. The first step is to construct a program representation that describes the control flow, data flow and data structures in the program. The details of this representation are described below. The second step is to analyze this representation and to suggest a number of possible program partitionings to the programmer, together with instructions on how to implement them (i.e. the code regions selected for execution on the accelerator and description of the data structures used).

In this work, we use profiling to construct control flow and data flow in the program. It is perfectly possible to construct control flow and data flow representations using static analysis. The trade-off between the two approaches is well-understood. While static analysis is exact, it is also conservative, meaning that some non-existing dependencies are listed in the program representation. Profile-based analysis, on the other hand, shows the correct set of dependencies for the profiled executions, but it may miss some dependencies if the profiling input data sets are incorrectly chosen. Note however that the choice between static analysis and profiling is orthogonal to this work: the contribution of this work is the way in which the program representation is analyzed and remains the same whether dependencies are constructed using static analysis or profiling.

2.1 Program Representation

In this section we introduce the necessary concepts on how to represent the control and data flow of a program to find sub-algorithms. These concepts are used throughout the rest of the paper.

In a program we identify three types of **code regions**: function bodies, loop bodies and general code fragments (snippets). Code regions are strictly nested, i.e. every code region is completely contained in a larger code region.

The code regions are strictly nested and are used as building blocks for our analysis to monitor the control and data flow. Figure 2(a) shows the different code regions of a small program[1]. The control flow between different code regions is represented in a context sensitive *call tree* (Figure 2(b)). Note that function Z occurs twice in the call tree because both instantiations have a different code region as parent.

A **call path** is a sequence of edges from the call tree C $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$. This path is from node v_1 to v_n.

Since C is a tree, each path from the root to a leaf-node is unique. Each node in the call tree is thus uniquely identified by its call path from the root. An example call tree is represented in Figure 2(b). In general the root node of a program is the *main* function.

A node w is called a **descendant** of a node v, if v is on the call path from the root to w.

A **subtree** $S(v)$ in the call tree C is the tree with as root node v and all its descendants. We select **sub-algorithms** from the subtrees based on their control and data flow characteristics (discussed in Section 2.2). Since subtrees can be nested we can find sub-algorithms in different granularities, i.e. different amounts of code are executed per invocation of the sub-algorithm. By selecting a sub-algorithm with a different nesting level in the call tree, one can tune the size of the sub-algorithm to memory latency and bandwidth and to the communication delay characteristics of particular accelerators. One of the contributions of this work is to limit the search for the optimal sub-algorithm to a small number of most interesting candidates. How interesting a subtree is depends on its data dependence properties.

A code region m is **data dependent** on code region n if it reads data produced by n. If its data is stored in data structure ds, we write $n \xrightarrow{ds} m$.

In Figure 3(a) we repeat the example from the previous section, but show besides the code regions also the data structures and their size. The solid arrows still represent the control flow, while the dotted arrows are the data dependencies. An arrow from a code region to data structure

[1]The functions *main*, X, Y and Z can also contain snippets, but these are omitted for clarity.

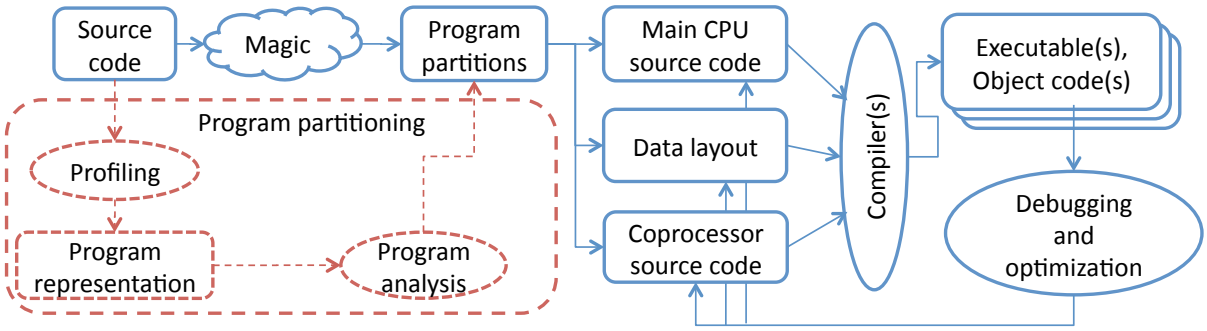
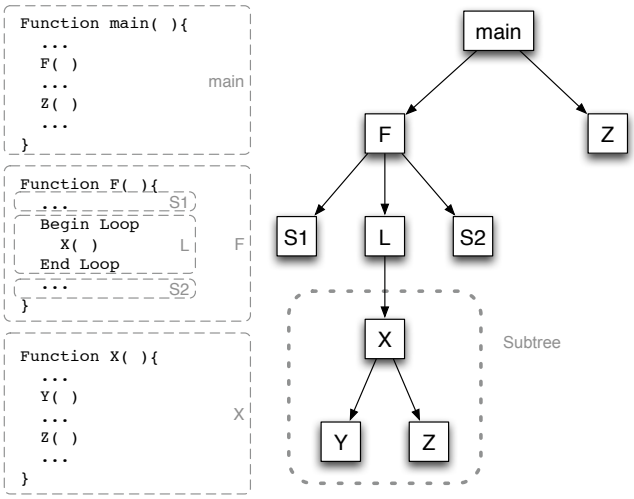


Figure 1: Work flow for program acceleration



(a) Source code of example (b) Call tree of example

Figure 2: An example to illustrate the terminology

means that this code region is writing information in it. An arrow from a data structure to a code region means that this code region is reading from this data structure. We see that X is data dependent on $S1$: $S1 \overset{ds2}{\rightarrow} X$. Note that this graph is related to the program dependence graph [9].

2.2 Program Partitioning

In this section we introduce three heuristics to find suitable partitions. In a first heuristic, we consider scenarios where a sub-algorithm makes strong use of private data structure, i.e. these data structures do not have to be copied between the main processor's memory and the accelerator memory at all. Due to the way programs are often structured, this heuristic does not always detect enough sub-algorithms. We present 2 more heuristics that use different constraints and hence detect more sub-algorithms.

2.2.1 Heuristic A: Based on Private Data

A first way of finding sub-algorithms is finding subtrees that have *private data*: data used only by the sub-algorithm, i.e. internal or temporary state. The idea behind this heuristic is that if a subtree has private data, it forms an independent entity within the program, having its own data structures for its specific task.

Definition 2.1. *A data structure ds is considered* ***private*** *to a subtree $S(v)$ iff*

$$\exists n, m \in S(v) \colon n \overset{ds}{\rightarrow} m$$

$$\forall n \in S(v) \colon \nexists m \in C(v) \setminus S(v) \colon (n \overset{ds}{\rightarrow} m) \vee (m \overset{ds}{\rightarrow} n)$$

Note that when a data structure *ds* is private to a subtree $S(v)$ it can still be used by other code regions not part of the subtree as long as there is no data dependency. So the data structures can be reused (a name dependence).

Each function that has local variables fulfills the heuristic of having private data, so each function is a sub-algorithm. However, we consider this as a trivial case and we do not call it a sub-algorithm. Also the main function is a special case, since almost all program data is private at that level. Moreover, we require that the subtree must have more private data than its children. Otherwise all the ascendants of a subtree with private data become defined as sub-algorithms.

Figure 3(b) marks the root of the detected sub-algorithms using this first heuristic in the graph with a grey background. By using this heuristic we detect that the subtree with root X has its own private data and as a result is a sub-algorithm. Figure 3(a) shows that subtree $S(X)$ is the only one using data structure *ds3*. The subtree of X also has shared data structures, namely *ds2* and *ds4*. The subtree $S(L)$ has no new private data, so it is not a sub-algorithm. However, subtree $S(F)$ has new private data besides *ds3* that was already private in subtree $S(X)$, namely *ds2* and *ds4*. Note that *ds*4 is also used outside the subtree, but this a name dependence (caused by a reuse of this data structure) which can be avoided by duplicating this data structure.

2.2.2 Heuristic B: Based on Shared Data

Depending on the programming style some programs make only little use of private data structures. Hence, the second heuristic for finding a sub-algorithm is comparing the *amount of shared data* of a subtree, with the amount of shared data structures of its parent subtree. If it has less shared data than its parent, it is a sub-algorithm.

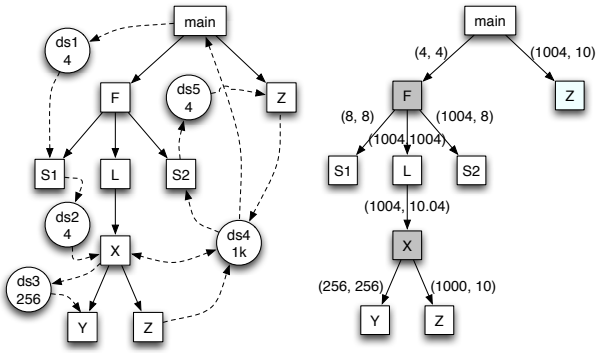
Definition 2.2. *A data structure ds is* ***shared*** *by a subtree $S(v)$ and the remainder of the program iff*

$$\exists n \in S(v) \colon \exists m \in C(v) \setminus S(v) \colon (n \overset{ds}{\rightarrow} m) \vee (m \overset{ds}{\rightarrow} n)$$

This is notated as $shared(ds, S(v))$. The set of shared data structures of a subtree $S(V)$ is defined by

$$shared(S(v)) = \{ds \colon shared(ds, S(v))\}$$

Definition 2.3. *The* ***amount of shared data*** *of subtree*



(a) Data usage of example (b) Call tree of example with annotaded edges

Figure 3: The root of each sub-algorithm has a grey background. The two weights on the edges in the figure on the right indicate the amount of shared data and the traffic for the underlying subtree

$S(v)$ is defined by

$$\sum_i sizeof(ds_i) \qquad with \quad ds_i \in shared(S(v))$$

The idea is similar to the first heuristic, but instead of finding subtrees that have more private data, we look for subtrees that have less shared data with the nodes outside the subtree. Normally the closer one gets to the root of the call tree (the main function), the less shared data one has, since in the root node the amount of shared data is zero. However, if we find a local minimum on the call path, this indicates an interesting subtree to cut off. A benefit of this heuristic is that we put no requirements on the existence of private data. In the evaluation (Section 3) this turns out to make a big difference in the number of detected sub-algorithms. Again we ignore the case of a single function (leaf nodes in the call tree) for our evaluation.

In Figure 3(b) the total amount of shared data for each subtree is indicated as the first number on the edges. One can make a difference between the amount of shared data that is read, written and the total amount. If the total amount is equal to the sum of the read and written shared data, it means the subtree has a separate read and write set. In order not to overload the figure we leave out the amount of read and written shared data and just show the total amount of shared data. In this simple example we find no subtree that meets our requirements. The reason is that we have a large data structure *ds4* that is shared by different nodes, giving a large value for the amount of shared data.

2.2.3 Heuristic C: Based on Data Traffic

The last heuristic for detecting sub-algorithms is by looking at data traffic of a subtree with the rest of the program. The data traffic of a subtree is based on the average amount of data that is read and written from shared data. If the traffic of a subtree is less than its parent, it is a sub-algorithm. The motivation for this heuristic is that it can find sub-algorithms that share large data structures with the outside world, but that only have few dependencies. In other words communication overhead is low, something that is disregarded by the second heuristic.

DEFINITION 2.4. ***Data traffic*** *for a subtree is defined by*

$$\frac{amount\ of\ read\ and\ written\ shared\ data\ of\ S(v)}{execution\ count\ of\ v}$$

In Figure 3(b) the data traffic is indicated by the second number on the edges. In this case we find that the subtree $S(X)$ is a sub-algorithm since its traffic is smaller than to that of its parent L. So where the second heuristic did not classify this subtree as a sub-algorithm because of the large amount of shared data, this heuristic shows that on the communication overhead is limited.

3. EVALUATION OF THE HEURISTICS

3.1 Experimental Setup

To evaluate our sub-algorithms, we consider 30 benchmarks that are a mixture of integer benchmarks from SPEC 2000 and the embedded MiBench suite [11]. Table 1 provides an overview of the benchmarks. We use a profiling tool [21] to obtain the dependence information of the programs needed to detect the sub-algorithms. All benchmarks are compiled with *gcc 4.1.0* for a powerPC 750 on linux.

3.2 The Number of Detected Sub-Algorithms

Figure 4 shows for each benchmark the number of sub-algorithms detected by the 3 heuristics and the total number of unique sub-algorithms. If identical subtrees are detected (they have the same code regions) they are only counted as one sub-algorithm. The main function, which represents the complete program, is not counted as a sub-algorithm. Also leaf nodes and subtrees with a library function as root are not taken into account. Moreover, the execution time of a subtree needs to be at least 1% of the total execution time.

The number of total detected sub-algorithms varies widely. For eight benchmarks only one or two sub-algorithms are detected. Nine benchmarks have at least 15 sub-algorithms. The limited number of sub-algorithms is typical for small embedded benchmarks in the MiBench suite. Spec CINT 2000 applications are much bigger and have a larger number of sub-algorithms.

For heuristic A, based on private data, we get the smallest number of sub-algorithms and in 11 cases we are not able to detect any sub-algorithm using this heuristic. This is caused by the fact that none of the subtrees have private data structures. So if the inner workings of a program are mostly based on shared data, this heuristic is unable to find sub-algorithms. When using the amount of shared data (heuristic B) or the data traffic (heuristic C), we get a higher number of sub-algorithms than heuristic A. Both heuristics also detect more sub-algorithms with a loop at the root of the subtree.

3.3 The Coverage of Detected Sub-Algorithms

Not only the number of sub-algorithms is important, also their coverage needs to be high for several of the potential applications of sub-algorithms. If the maximum coverage of the detected sub-algorithms only comprises a small fraction of the total execution time, they are not suitable for off-loading or representative for performance evaluation. However, Figure 5 shows that in most cases the combined heuristics for detecting sub-algorithms have a coverage of more than 99%. The coverage is less than 50% for *qsort*, *lame* and *gsm*.

Table 1: Benchmarks used in this study along with their inputs and dynamic instruction counts (in millions)

	Benchmark	Input	Cnt(M)		Benchmark	Input	Cnt(M)
SPEC CPU2000				MiBench			
CINT	bzip2	train	62547	office	ispell	small	9
	gzip	train	43141		sphinx	small	2062
	mcf	train	7658		stringsearch	small	0.2
MiBench				security	blowfish.dec	small	81
automotive	basicmath	small	64		blowfish.enc	small	81
	bitcount	small	34		pgp.sign	small	26
	qsort	small	43		rijndael.dec	small	29
	susan.corners	small	1		rijndael.enc	small	30
	susan.edges	small	2	telecomm	sha	small	12
	susan.smoothing	small	35		adpcm.dec	small	25
cons	jpeg.dec	small	5		adpcm.enc	small	30
	jpeg.enc	small	23		crc32	small	112
	lame	small	118		fft.inv	small	41
netw	dijkstra	small	55		fft	small	37
	patricia	small	87		gsm.dec	small	22
					gsm.enc	small	53

The maximum coverage of sub-algorithms detected by heuristic A (if any) is comparable to the coverage of heuristic B & C. In some benchmarks (e.g. *blowfish* and *fft*) the best coverage is achieved by the heuristic of data traffic (heuristic C).

3.4 Comparison of the Heuristics

From the previous evaluation we know that the three heuristics have a high coverage. Another important aspect is the overlap of the detected sub-algorithms between the different heuristics. This is shown in Figure 6. Each letter represents the corresponding heuristic that detects a specific sub-algorithm. So *AB* means the a sub-algorithm is detected by both heuristic A and B, while *ABC* means that the three heuristics find this sub-algorithm.

Both A, B and C have fractions in the graph, meaning that each heuristic finds unique sub-algorithms not detected by the other two heuristics. However, only heuristic C has a reasonable amount of unique sub-algorithms compared to the other two heuristics. The largest fraction in the graph is BC. So the results from shared data (heuristic B) and data traffic (heuristic C) are most closely related to each other.

4. PROOF OF CONCEPT

In Section 4.1 we give an in-depth analysis of the detected sub-algorithms in bzip2 and in Section 4.2 for mcf. In Section 4.3 we evaluate the performance of a sub-algorithm run on the SPEs of a Cell processor.

4.1 Case Study of Bzip2

Figure 7 shows the call tree of the major code regions of the compression part of bzip2. Basically the program has a compression (spec_compress) and a decompression (spec_uncompress) routine which both consists of a loop that performs the necessary encoding or decoding steps. We will mainly focus on the compression part, since this is the more time-consuming part of the program.

Sub-algorithms based on private data.

Each of the code regions, except for the three nodes with a dotted border (*loadAndRLEsource*, *getRLEpair* and *fullGtU*) in Figure 7 are sub-algorithms. In Table 2 we summarized the number of private and shared data structures for each subtree[2]. This explains immediately why *loadAndRLEsource* and *getRLEpair* are not detected as a sub-algorithm: they do not have any private data structures.

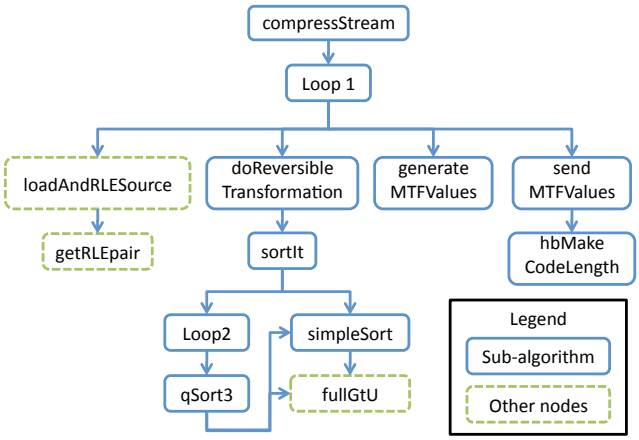


Figure 7: High-level overview the most important call paths within the compression part of bzip2

Table 2 also shows the coverage of execution time for each subtree compared to the total execution time. We see that the compression part of bzip2 is responsible for about 86.6%, while the decompression for 13.4%. Also note that the majority of the execution time of *doReversibleTransformation* is spent in the sub-algorithm *sortIt*.

Sub-algorithms based on shared data.

In Figure 8 we show for different call paths the amount of shared data. Each line in the graph represents a different call path of the call tree in Figure 7 for the compression

[2]Note that in practice our tool gives the name of the involved data structures, however, due to space restrictions we just show the number of data structures.

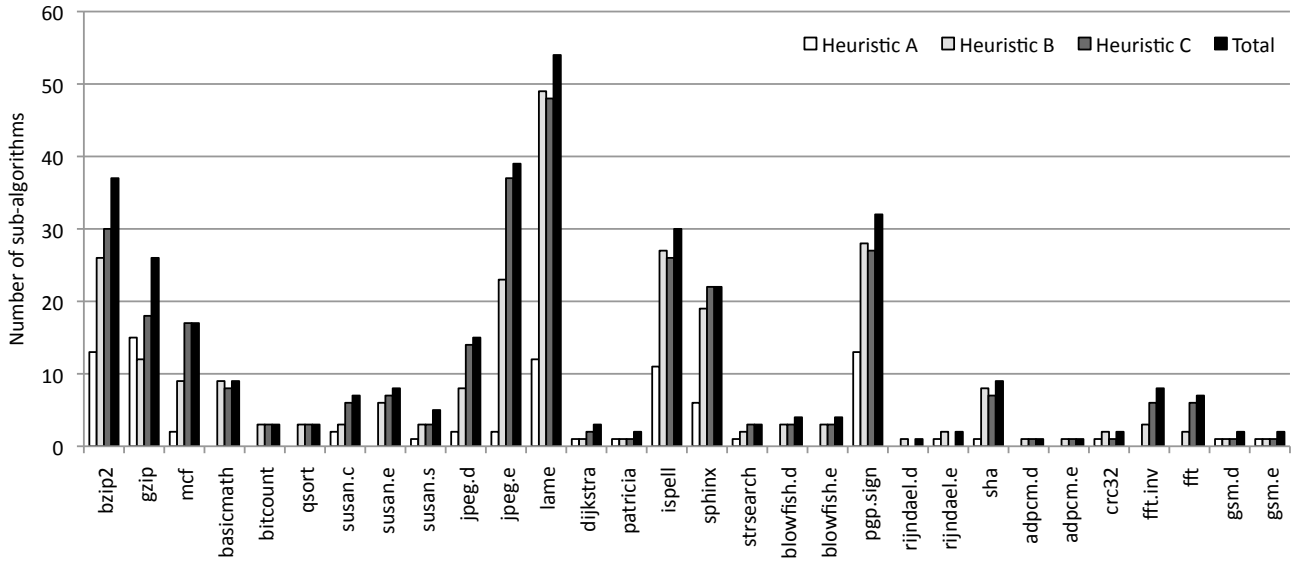


Figure 4: Number of detected sub-algorithms for each heuristic

Table 2: Information on number of private and shared data structures and coverage for different subtrees of bzip2

Root of subtree	Cov (%)	Prv DS	Shr DS
spec_compress	86.59	32	15
compressStream	86.59	31	16
Loop1	86.59	23	22
loadAndRLEsource	5.32	0	11
getRLEpair	4.60	0	6
doReversibleTf	59.24	6	15
sortIt	58.91	2	13
generateMTFValues	16.39	1	10
sendMTFValues	5.64	5	15
hbMakeCodeLenghts	3.80	1	6
spec_uncompress	13.40	28	21
uncompressStream	13.40	27	22
Loop2	13.40	20	15
getAndMoveToFrontDec	11.40	10	15
undoReversibleTf	2.00	2	15

part. Reading the X-axis from left to right is equivalent to traversing the call tree from a leaf node up to the main node of the program. The amount of shared data in *main* is of course zero, since this is the root node of the call tree.

As opposed to the first heuristic, the second heuristic does not identify the subtrees higher in the call tree (*Loop1*, *compressStream* and *spec_compress*) as sub-algorithms since they have no smaller amount of shared data compared to their parent. The subtrees beneath *Loop1*, however, are identified as sub-algorithms. Now even *loadAndRLEsource* and *getRLEpair* are marked as such, as the second heuristic poses no requirements on the existence of private data. The analysis also detected other sub-algorithms that are incorporated in the subtrees shown here. However, since their coverage is much smaller they are not shown.

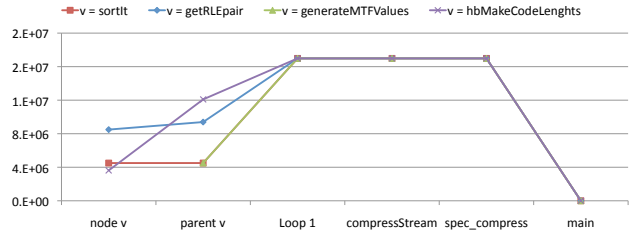


Figure 8: Shared data for different call paths of bzip2

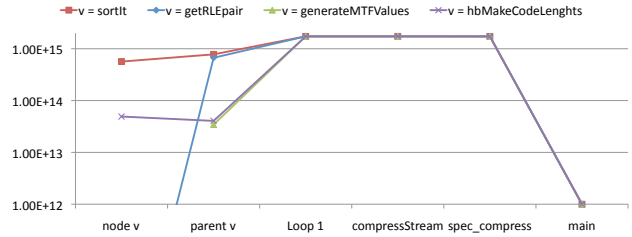


Figure 9: Data traffic for different call paths of bzip2

Sub-algorithms based on data traffic.

The results for the third heuristic, based on data traffic, are shown in Figure 9. While this graph shows a lot of similarities with Figure 8 from the second heuristic, there are some important but subtle distinctions. The subtree of *hbMakeCodeLengths* is identified as a sub-algorithm based on the amount of shared data . However, based on the data traffic, we see this is no longer the case. The data traffic of its parent subtree, *sendMTFValues*, is actually smaller. So communication-wise, *hbMakeCodeLengths* is no longer an interesting sub-algorithm. The identification of *getRLEpair* as a sub-algorithm by the second heuristic, however, is empha-

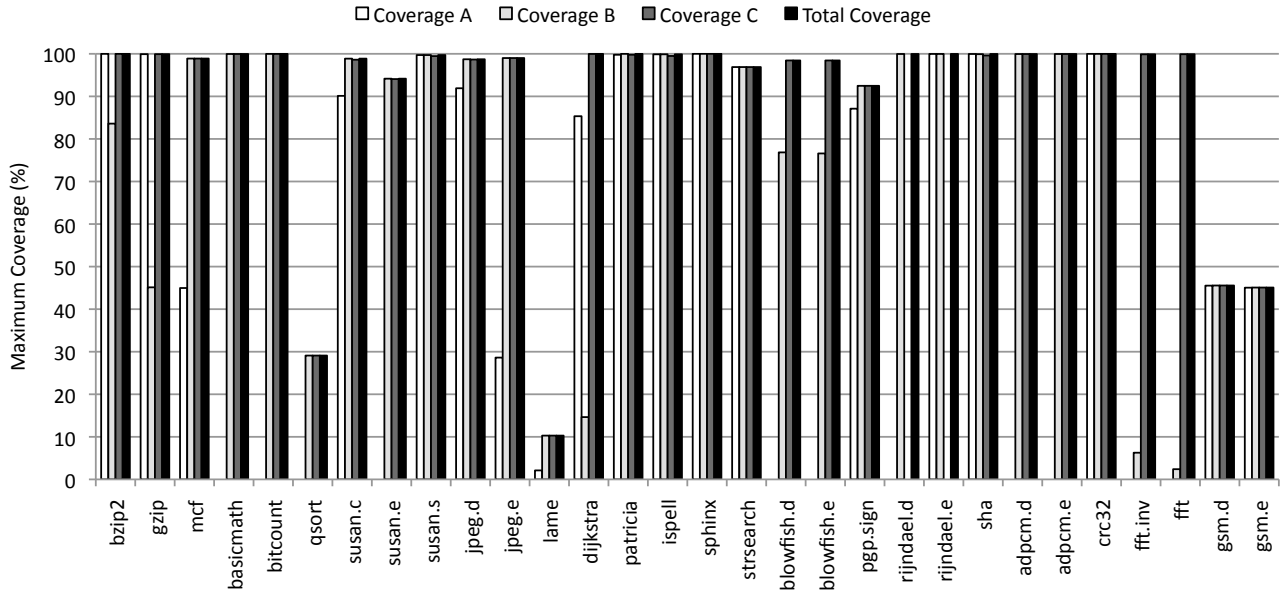


Figure 5: Coverage of detected sub-algorithms for each heuristic in percentage of total execution time

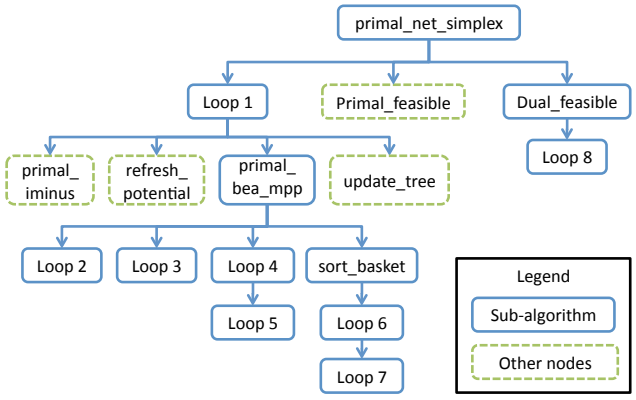


Figure 10: High-level overview the most important call paths within the primal_net_simplex of mcf

sized by the third heuristic. Based on the amount of shared data it already showed a small benefit, but if we look at the data traffic we find a huge advantage (note that the point lies beneath the scale we used).

4.2 Case Study of Mcf

The mcf benchmark does some vehicle scheduling which is formulated as a large-scale minimum-cost flow problem. This is solved by using a network simplex algorithm. Hence three quarters of the time is spend in *primal_net_simplex*. In Figure 10 we show an overview of the call paths of this part.

Sub-algorithms based on private data.

In Table 3 we see that most subtrees have no private data structures. As a result only *read_min* (used for reading the input) and *primal_bea_mpp* are identified as sub-algorithms with our heuristic based on private data. The parents of *primal_bea_mpp* (*Loop1* and *primal_net_simplex*) have no extra private data compared to their child, so they are not classified as sub-algorithms by the first heuristic. It is clear that since the algorithm in mcf is based on a few data structures that are shared throughout the entire program that the first heuristic cannot find many sub-algorithms.

Sub-algorithms based on shared data.

The second heuristic compares the amount of shared data of a subtree to the amount of shared data of its parent. This information is shown in the column *Shared ratio* of Table 3: if the ratio is > 1 it means that the parent has the same or more shared data (good), if it is ≤ 1 the subtree has more shared data than its parent (bad). Although we find more sub-algorithms using this heuristic, we see that in most cases that ratio is close to one $(1+\varepsilon)$. The reason for the marginal improvement in amount of shared data is the fact that the largest shared data structures are used everywhere. Only for *sort_basket* there is a real improvement, because it does not use the larger shared data structures.

Sub-algorithms based on data traffic.

For the third heuristic we can use the last column (*Comm ratio* in Table 3. Again a ratio bigger than one means the subtree has on average less traffic compared to the communication of its parent. This time we see a much better result. Most of the detected sub-algorithms even have a significant reduction in data traffic. This indicates that while the data flow in mcf is mainly comprised by a few large shared data structures, on average only a small amount of that data is used for each invocation.

4.3 Acceleration on the Cell BE

The evaluation of the acceleration is done on a PlayStation 3 running Linux kernel 2.6.23 and is compiled with *gcc.4.1.2*. The most important characteristics of this processor are provided in Table 4. We only made an implementation for two benchmarks because getting a lot of performance out of the SPEs on a Cell BE is a very time consuming process that requires a lot of hand-tuning.

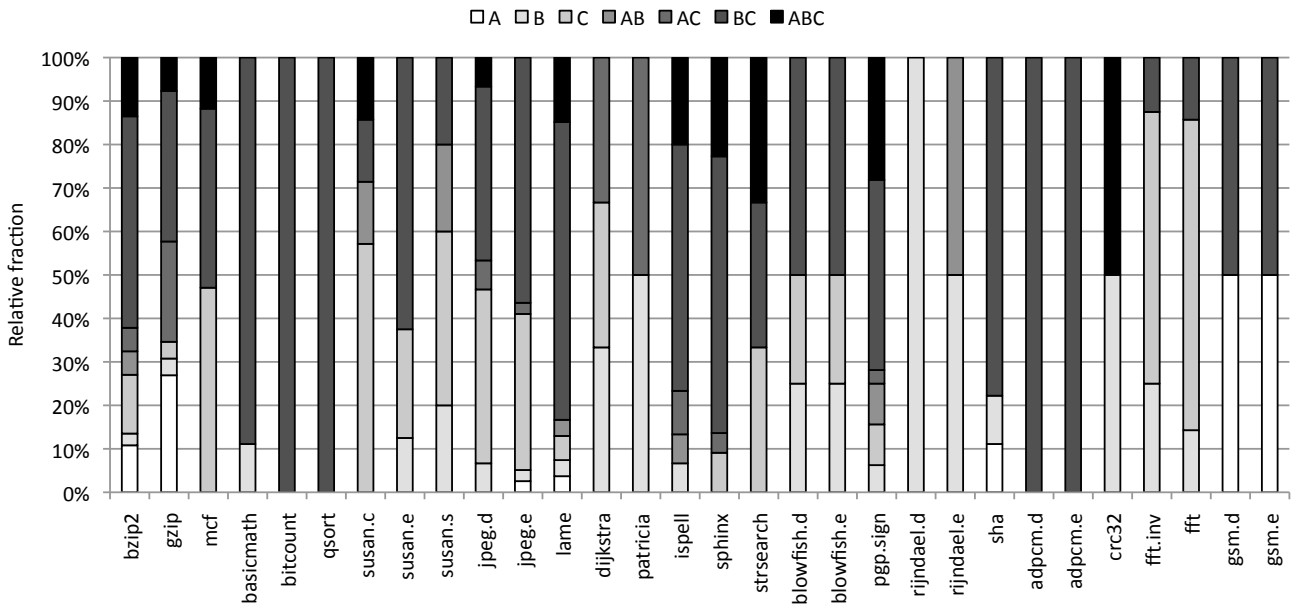


Figure 6: Breakdown of different sub-algorithm detection mechanisms

Table 3: Information on coverage, number of private and shared data structures, amount of shared data and communication compared to their parent for different sub-algorithms of mcf

Root of subtree	Crit	Coverage (%)	Private DS	Shared DS	Shared ratio	Comm ratio
read_min	ABC	3.63	2	40	$1+\varepsilon$	219.0
price_out_impl	BC	27.69	0	3	1.004	78611.1
primal_net_simplex	BC	67.57	5	5	$1+\varepsilon$	8297.2
Loop1	C	66.30	5	5	1	1.2
primal_bea_mpp	ABC	41.34	5	3	1.004	90.4
Loop3	C	1.41	0	2	1	11.4
Loop4	C	23.23	0	7	0.99	1.1
Loop5	BC	22.03	0	4	$1+\varepsilon$	7.7
sort_basket	BC	13.87	0	2	15147	8.1
Loop6	C	10.26	0	2	1	1.5
Loop7	C	1.72	0	2	1	9.3
dual_feasible	BC	1.22	0	3	1.004	791.2
Loop8	C	1.16	0	3	1	$1+\varepsilon$

4.3.1 Running Bzip2 on the Cell BE

The part that is accelerated on the SPEs is the sub-algorithm *simpleSort* that performs a shell's sort. This part takes about 20% of the execution time and calls upon a variable-length-compare (*fullGtU*). The sub-algorithm *simpleSort* is called by a quickSort algorithm (*qSort3*) and the main sorting routine *sortIt*. Both *simpleSort* and *fullGtU* are run on the SPEs. Moreover, there is parallelism between the calls from *qSort3* to *simpleSort*, allowing to use several SPEs. The reason for choosing *simpleSort* as the sub-algorithm to accelerate is that it takes most of the execution time of the compression part. In Figure 7 we see that its parents are also defined as sub-algorithms. However, trying to accelerate them on the Cell BE will be problematic due to limited size of the local store of an SPE. As explained before, one has to pick the proper granularity for the desired accelerator. In this case the optimum point is in *simpleSort*. Its parent *sortIt* has a very control intensive part and should be left on the PPE side.

If we use hot code analysis, we will find that *fullGtU* is the hottest region in this part of the program. However, it is completely data dependent on information provided by *qSort3* and *simpleSort*. Hence, off-loading only *fullGtU* solely based on its hotness would result in a large communication overhead. Our heuristics, however, clearly show that *fullGtU* has no private data structures (heuristic A), too much shared data (heuristic B) and also has an unfavorable amount of communication (heuristic C).

The speedup results are shown in Figure 11(a). The *qSort3* runs 56% faster allowing the compression part to finish 14% times faster. This results in a total speedup of 9%.

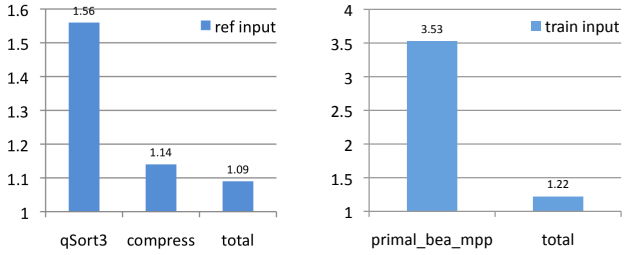
4.3.2 Running Mcf on the Cell BE

For accelerating mcf we move the entire *primal_bea_mpp* sub-algorithm to the SPUs. This sub-algorithm takes 41.3% of the execution time and consists of several nested loops and *sort_basket*. The main loop of *primal_bea_mpp* is *Loop4* and can be parallelized speculatively across several SPUs.

Looking at the hottest loops in *primal_net_simplex* brings up *refresh_potential* (Figure 10) that is responsible for about

Table 4: Cell specifications for PlayStation 3

	1 PowerPC Processor Element (PPE)	6 Synergistic Processor Elements (SPEs)
Type of processor	64-bit in-order RISC @ 3.2 GHz	128 bit in-order vector processor
Memory hierarchy	32 KB L1 data and instruction cache and unified 512 KB L2 cache	256 KB local storage
Properties	two-way simultaneous multithreading two-way superscalar	no hardware branch prediction, explicit memory management



(a) bzip2 (b) mcf

Figure 11: Speedup results using 6 SPEs as accelerators compared to single PPE execution

20% of the execution time. However, its data communication characteristics are very unfavorable, making this a bad choice for off-loading to an accelerator.

The speedup results of *primal_bea_mpp* are shown in Figure 11(b). The sub-algorithm runs 3.53 times faster than the original version on the PPE. In total mcf runs 22% times faster thanks to the accelerators.

5. RELATED APPLICATION FIELDS FOR SUB-ALGORITHMS

Based on our evaluation sub-algorithms appear to be well-suited entities for off-loading functionality on acceleration cores, e.g. off-loading to the SPUs in the Cell BE processor [20] or to a GPU. In general, function off-loading can be considered a special case of program partitioning, which is typically formulated as a min-cut flow problem [8]. A related problem was considered for Multiscalar processors [24], where tasks are identified with minimum communication cost. The size of these tasks is, however, limited. Sub-algorithms on the other hand are found in different granularities, but in most cases they will be too big to be suitable for the Multiscalar.

Performance evaluation is another area where sub-algorithms can contribute. Analyzing and simulating large programs is a time consuming job for computer architects. Furthermore, it is important that analysis and simulation are reproducible. Hence, performance evaluation is typically performed using small and manageable benchmarks. The construction of such benchmarks is not algorithmically described in the literature but it is a rather ad hoc procedure [14, 17]. We believe that sub-algorithms are a viable first step towards the automatic identification of pieces of programs that are usable as stand-alone benchmarks by extracting sub-algorithms from real-life programs. An alternative approach discussed in the literature is to reduce large applications to synthetic benchmarks that exhibit the same architectural behavior, but do not have any real functionality [2]. Such heuristics, however, are restricted to computer architecture evaluation. They cannot be used, e.g., to evaluate compiler technology.

The relative cost for maintaining software and managing its evolution now represents more than 90% of its total cost. This is referred to as the *legacy crisis* by Seacord et al. [23]. Hence, program understanding becomes an important field in software development. It is the software engineering discipline concerned with understanding existing programs with the goal of facilitating code maintenance. Hereto, one tries to find a mapping between features of programs (which can be the functionality, requirements or other concerns of the program) to the actual source code. Many of these heuristics rely, amongst others, on execution-driven analysis in order to identify the code regions that are executed when a particular concern is exercised [5, 7]. Also, techniques have been developed to visualize run-time dependencies between features [16]. These elements are also present in our analysis, i.e. in the profiling information and in the graph representation of programs. Furthermore, we believe that the sub-algorithms may provide additional benefits in describing the structure of a program in relation to the usage of data structures, which may give programmers additional insight before delving into the source code.

For certain input parameters a function can be further optimized, a technique known as code specialization. Previous research [1] studied different techniques to specialize a C-program using both analysis of control and data flow. Instead of using functions for specialization one can also consider using sub-algorithms. By using value profiling it is possible to find common cases of sub-algorithm that is suitable for optimization.

6. CONCLUSION AND FUTURE WORK

We have presented a technique to extract functionality from a program, so called *sub-algorithms*, that forms an isolated entity within the program and that is independent of the instruction set. We introduced three heuristics to detect such sub-algorithms, all of them focused on data usage. The first heuristic searches for an increase in private data, the second for a decrease in the amount of shared data and the third in a decrease of the data traffic.

Our experimental evaluation shows for thirty benchmarks that we are able to detect tens of these sub-algorithms even in small programs. In most cases several sub-algorithms in a program are nested. This allows one to chose the suitable granularity for the desired architecture. We used sub-algorithms as accelerator on the Cell BE, resulting in good speedups even for programs with complex control flow.

In this work we mainly focussed on detecting sub-algorithms for acceleration. A next step would be to provide this information to a compiler in order to further automate the process of program partitioning.

Acknowledgments

Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral research fellow with the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] R. H. Bell. Automatic workload synthesis for early design studies and performance model validation. *lib.utexas.edu*, page 169, 2005.

[3] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSS: a programming model for the Cell BE architecture. In *SC '06*, page 86, 2006.

[4] A. Cantle and R. Bruce. An Introduction to the Nallatech Slipstream FSB-FPGA Accelerator Module for Intel Platforms. White paper, http://www.nallatech.com, Sept. 2007.

[5] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC 2008: The 16th IEEE International Conference on Program Comprehension*, pages 53–62, June 2008.

[6] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the Cell processor. In *PACT '05*, pages 161–172, 2005.

[7] A. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *ICSM'05*, pages 337–346, Sept. 2005.

[8] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *Information Theory, IEEE Transactions on*, 2(4):117–119, 1956.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[10] a. Gabriel Falc L. Sousa, and V. Silva. Massive parallel LDPC decoding on GPU. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 83–90, 2008.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Intl. Workshop on Workload Characterization*, 2001.

[12] T. R. Halfhill. Floating point buoys ClearSpeed. *Microprocessor Report*, page 7, Nov. 2003.

[13] IBM. Performance Analysis with the IBM Full-System Simulator. Documentation, http://www.ibm.com/developerworks/power/cell/, Sept. 2007.

[14] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[15] J. H. Kelm, I. Gelado, M. J. Murphy, N. Navarro, S. Lumetta, and W. mei Hwu. CIGAR: Application partitioning for a CPU/coprocessor architecture. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 317–326, 2007.

[16] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 59–68, 2007.

[17] D. J. Lilja. *Measuring Computer Performance*. Cambridge University Press, 2000.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[19] T. Mattson. Introduction to openmp. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 209, 2006.

[20] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation Cell processor. In *ISSCC 2005, IEEE International Solid-State Circuits Conference*, pages 184–592, 2005.

[21] S. Rul, H. Vandierendonck, and K. De Bosschere. Detecting the existence of coarse-grain parallelism in general-purpose programs. In *Proceedings of the First Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG-1*, page 12, 1 2008.

[22] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09*, pages 131–140, 2009.

[23] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[24] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[25] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 189–198, 2003.

[26] H. Vandierendonck, S. Rul, M. Questier, and K. De Bosschere. Experiences with parallelizing a bio-informatics program on the Cell BE. In *HiPEAC 2008*, volume 4917, pages 161–175. Springer, 1 2008.

[27] O. Villa, D. P. Scarpazza, and F. Petrini. Accelerating real-time string searching with multicore processors. *Computer*, 41(4):42–50, 2008.

[28] D. H. Woo, H.-H. S. Lee, J. B. Fryman, A. D. Knies, and M. Eng. Pod: A 3D-integrated broad-purpose acceleration layer. *IEEE Micro*, 28(4):28–40, 2008.