Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications

Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili

School of Electrical and Computer Engineering

Georgia Institute of Technology

Atlanta, GA, USA

Running head: Control Flow Transformation

Haicheng Wu (Corresponding Author)

E-mail: hwu36@gatech.edu

Gregory Diamos

E-mail: gregory.diamos@gatech.edu

Jin Wang

E-mail: jin.wang@gatech.edu

Si Li

E-mail: sli@gatech.edu

Sudhakar Yalamanchili

E-mail: sudha@gatech.edu

Address:

266 Ferst Drive, KACB 2316

Atlanta, GA 30332-0765

Phone: (404)894-2940

Abstract

This paper identifies important classes of program control flow~~s~~ in applications targeted to ~~commodity~~ commercially available graphics processing units (GPUs) and characterizes their presence in real workloads such as those that occur in CUDA and OpenCL. Broadly, control flow can be characterized as structured or unstructured. It is shown that most existing techniques for handling divergent control in bulk synchronous GPU applications handle structured control flow efficiently, some are incapable of executing unstructured control flow directly, and none handles unstructured control flow efficiently. An approach to reduce the impact of this problem is provided.

An unstructured-to-structured control flow transformation for CUDA kernels is implemented and its performance impact on a large class of GPU applications is assessed. The results quantify the importance of improving support for programs with unstructured control flow on GPUs. The transformation can also be used in a JIT compiler pass to execute programs with unstructured control flow on the GPU devices that do not support unstructured control flow. This is an important capability for execution portability of applications using GPU accelerators.

Keywords: GPU, Unstructured Control Flow, Branch Divergence

1 Introduction

The transition to many-core computing has been accompanied by the emergence of heterogeneous architectures driven in large part by the major improvements in ~~j~~Joules/operation and further influenced by the evolution to throughput-oriented computing. This has coincided with the growth of data parallel computation that has become a pervasive and powerful model of computation. ~~whose~~ Its importance has been amplified by the rate at which raw data is being generated today in all sectors of the economy and rapidly growing in the foreseeable future. The emergence of low-cost programmable GPU computing substrates from NVIDIA, Intel, and AMD have made data parallel architectures ~~commodity~~commercially available from embedded systems through large scale clusters such as the Tsubame (Matsuoka 2008) and Keeneland systems (NICS 2010), hosting thousands of NVIDIA Fermi chips. Major research foci now include the development of programming models, algorithms, applications, performance analysis tools, productivity tools, and system software stacks.

Emerging data-parallel languages that implement *single instruction stream multiple thread* (SIMT) models (Rixner et al. 1998) such as CUDA and OpenCL retain many of the control flow abstractions found in modern high level languages and simplify the task of programming these architectures. However, when the SIMT threads do not follow the same control path, performance suffers through poor hardware utilization and dynamic code expansion. This problem of *branch divergence* is critical to high performance and has attracted hardware and software support. The impact of branch divergence can be quite different depending ~~up~~on whether the program's control flow is structured (control blocks have single entry and single exit such as if-then-else) or unstructured (control blocks have multiple entries or exits such as those using *goto* statements). In fact, some GPUs will only support (and hence their compilers will only generate) structured control flow. Therefore it becomes important to understand the impact of unstructured control flow in GPU applications and performance effects of techniques developed to deal with it. This understanding is critical to the development of new techniques to improve the efficiency of support for unstructured control flow. This in turn can lead to the support of advanced features in GPGPU programming ~~such as try/catch~~ that produces unstructured control flow and are currently not supported in GPU architectures , such as try/catch.

A second reason for understanding the impact of unstructured control flow and the development of supporting compiler technology is the emerging importance of portability in future heterogeneous many core architectures. Chips that support multiple instructions on a die, such as Intel's Sandybridge (AVX, x86, GEN), will be common. Execution portability can be achieved via dynamic translation to support multiple GPU back-ends (Diamos et al. 2010). The ability to execute a GPU kernel on multiple targets enhances portability and protects software investments. For example, transformations between unstructured and structured control flow implementations are necessary when one of the GPUs does not natively support unstructured control flow, e.g., AMD Radeon (Dominguez et al. 2011). In reality, there already exists programs such as Optix (Parker et al. 2010) that have complex unstructured control flow and need to be accelerated by GPUs. The current limited bandwidth between GPUs and CPUs forbids lots of data movement between them when running these programs in a high throughput system and makes the support of unstructured control flow on GPUs a desirable solution in these cases. The future change of the balance between the bandwidth and the accelerator complexity may alter the design decision, but it remains to see.

In this paper we seek to analyze the occurrence and impact of unstructured control flow in GPU kernels. This paper makes the following contributions:

- Assesses the occurrence of unstructured control flow in several GPU benchmark suites.
- Establishes that unstructured control flow necessarily causes dynamic and static code expansion for state of the art hardware and compiler schemes. It shows that this code expansion can degrade performance in cases that ~~do~~ occur in real applications.
- Implements a compiler intermediate representation (IR) ~~level~~ transformation ~~which~~ that can ~~transform~~ turn unstructured control flow to a structured control flow implementation. This transformation is useful for researching the performance of arbitrary control flow~~s~~ on GPUs, and is also important for execution portability via dynamic translation.

The rest of the paper is organized as follows: Section 2 introduces unstructured control flow and its specific manifestation~~s~~ in GPU codes. Section 3 describes transformations for converting

unstructured control flow to structured control flow. The experimental evaluation section, Section 4, assesses the impact of the transformations on several benchmark suites. Section 5 introduces the related work of this paper. The paper concludes with some general observations and directions for future work.

2 GPU Control Flow Support

Compilers can translate high level imperative languages such as C/C++ or Java into an intermediate representation (IR) that resembles a low level instruction set. Typical examples of IR are LLVM (Lattner and Adve 2004) , PTX for CUDA GPU (NVIDIA 2009), or AMD IL for AMD GPU (AMD 2009). In ~~the~~ IR ~~level~~, ~~the~~ Control Flow Graph (CFG) represents the execution path of the program. Every node of the graph is a group of sequentially executed instructions, and the edges are the jumps which are usually caused by conditional/unconditional branches.

Previous work (Zhang and H. D'Hollander 2004) classifies control flow patterns into two categories, structured and unstructured. In general, commonly used control flow patterns, such as those shown in Figure 1, are **structured**. These patterns correspond to **hammock** graphs in the CFG which are defined as subgraphs having a single entry node and a single exit node (Ferrante et al. 1987). On the contrary, **unstructured** control flow may have multiple entries or exits. Figure 2 adds some extra edges (dot line), which may be caused by *goto* statements, to the structured control flow in Figure 1 and turns them into unstructured control flow. Based on the classification, Zhang et al. introduced a generic approach to transform graphs with unstructured control flow to graphs possessing structured control flow. This transformation will be explained in section 3. The remainder of this section introduces the common sources of unstructured control flow~~s~~ and how they are supported in SIMT architectures.

2.1 Sources of Unstructured Control Flow

One of the most common sources is the *goto* statement used in C/C++ which allows control flow to jump to arbitrary nodes in the CFG. Similarly, longjumps and exceptions are two other sources of

unstructured control flow.

However, even if the programming language forbids the use of *goto* statements (such as OpenCL), the compiler may also produce unstructured control flow in ~~the~~ IR ~~level~~ due to unintended side-effects of the language semantics. For example, in the code segment of Figure 3(a), ~~the~~ compiler does not need to evaluate all four conditions (which is known as a short-circuit optimization) and the CFG of the generated IR looks like Figure 3(b). This CFG has unstructured control flow because subgraph {B1, B2} and {B3, B4} both have two exits.

Moreover, CFG optimizations performed by compilers ~~would~~ can also cause unstructured control flow (Cooper et al. 2001). Considering Figure 4(a), if function *foo()* is inlined into the *main()* function, the early return statement in loop2 ~~would~~ will create the second exit from the loop, which is shown in Figure 4(b).

The first example highlights the difficulty in designing language semantics that ~~do not require~~ require non-existence of unstructured control flow, while the second example shows ~~demonstrates~~ that only a subset of existing compiler optimizations preserve the structured property of CFGs. As a result, designers of compilers for SIMD processors without hardware support for unstructured control flow must decide between performing existing optimization passes and then restructuring the program or avoiding certain optimizations altogether. This greatly increases the complexity of these compilers and potentially eliminates opportunities for optimization. Since the above examples are very common in modern programming languages, normal programs usually have both ~~the~~ structured and unstructured control flows. ~~If the system cannot execute unstructured parts efficiently, it would hurt the overall performance. Also ideally, it would be desirable to not place restrictions on language semantics and retain the ability to perform arbitrary transformations on CFGs. These requirements create a clear advantage to supporting unstructured control flow on general purpose SIMD processors.~~ The desires that unstructured parts should be executed efficiently to avoid hurting overall performance while not placing restrictions on language semantics and retaining the ability to perform arbitrary transformations on CFGs creates a clear advantage to support unstructured control flow on general purpose SIMD processors.

~~The above examples also show that, essentially, it is the interacting edge that causes the unstructured control flows. These interacting edges occur in two mutually exclusive cases:~~

In the above examples, if some edges are deleted, the control flow will become structured. These edges are called *interacting edges* (Zhang and H. D'Hollander 2004), since they interact with two ~~structured control flow~~hammock graphs. There are two types of such edges:

- an **interacting out-edge** leaves a hammock graph from a point other than the exit block, such as edge E1 and E2 in Figure 3(b). Edge E1 in Figure 4 is also an interacting out-edge.

- an **interacting in-edge** enters a hammock graph from a point other than the entry block. ~~Edge E1 in Figure 9(a) is an example~~The dotted line in Figure 2(b) and Figure 2(d) are two examples.

2.2 Impact of Branch Divergence in Modern GPUs

Modern programmable GPUs implement massively data parallel execution models. In this paper we analyze GPU kernels from CUDA applications compiled to NVIDIA's parallel thread execution (PTX) virtual instruction set architecture. PTX defines an execution model (see Figure 5) where an entire application is composed of a series of multi-threaded kernels. Kernels are composed of parallel work-units called *Cooperative Thread Arrays* (CTAs), each of which can be executed in any order subject to an implicit barrier between kernel launches. Threads within a CTA are grouped together into logical units known as *warps* that are mapped to *single instruction stream multiple data* (SIMD) units using a combination of hardware support for predication, a thread context stack, and compiler support for identifying re-converge points at control-independent code.

Since threads within the same *warp* have to execute the same instructions, branch control flow can potentially cause inefficiencies if the branch condition is not evaluated identically across all threads in a *warp*. In this case, some threads may take a fall-through edge and the others may jump to the branch target, which is referred to as branch divergence. This can be handled by a process of serially enabling/disabling threads corresponding to the then/else branch. This effectively splits the *warp* into smaller subsets of threads which may then re-converge later in the execution. The execution model of other GPUs are similar, though they use different terminology.

The implementation details of re-convergence differ among GPUs. In AMD GPUs illustrated in Figure 6, its IR language (AMD IL) uses explicit instructions such as IF, ELSE, ENDIF, LOOP, ENDLOOP, etc., which means it only supports limited structured control flow~~s~~ (AMD 2010). The mapping of these control flow~~s~~ to the hardware is simple and fixed. It executes all the possible paths of the program (e.g., *then* part and *else* part for IF instructions) in a lock-step manner, and threads re-converge at the END instructions such as ENDIF or ENDLOOP. If the compound condition code in Figure 3(a) is compiled for AMD GPUs, it has to generate CFG like ~~Figure 10(c)~~Figure 3(c) which uses nested if-then-else to form a structured control flow implementation. The Intel GEN5 graphics processors work in a similar manner (Intel 2009).

However, mapping parallel programs with arbitrary control flow~~s~~ onto SIMD units is a difficult problem because there is generally no guarantee that different parallel threads will ever be executing the same instructions. Thus, the re-convergence point may impact the overall performance. This will be discussed in the following subsection.

2.3 Unstructured Control Flow on GPUs

Although supporting structured control flow is sufficient for many graphics shading languages such as Microsoft DirectX and Khronos OpenGL, the migration to general purpose models such as OpenCL and CUDA that derive from C makes it advantageous to support unstructured control flow. Specifically, CUDA supports *goto* statements in the high level language. In addition, its IR language, PTX, has many features in common with RISC ISAs, which includes arbitrary branch instructions rather than explicit IF and LOOP instructions. Consequently, as discussed in Section 2.1, compilation of CUDA programs can employ common CFG optimizations that are already widely used in other C/C++ program compilation frameworks, and programmers do not need to worry about introducing unstructured control flow into programs that are not allowed on some GPU platforms.

The current state of the practice in determining re-convergence points for divergent SIMD threads is referred to as immediate post-dominator[1] re-convergence (Fung et al. 2007)~~ (the immediate post-dominator~~

[1] The immediate post-dominator of a branch in a CFG, informally, is the node through which all paths from the branch pass and which does not post-dominate any other post dominator.

~~of a branch in a CFG, informally, is the node through which all paths from the branch pass through and which does not post-dominate any other post dominator)~~. By using this method, the re-converge point is fixed for every divergent branch and can be calculated statically during compilation. For structured control flow, this method would re-converge at the end of loops or if-else-endif control blocks, which are as efficient as AMD GPUs. However, it may execute inefficiently for unstructured control flow. For example, in Figure 7, assume the *warp* size is 7 and these 7 threads take 7 different paths as shown in Figure 7(b), which is the worst case for this CFG. The immediate post-dominator of all branches is the exit node (see Figure 7(a)). Figure 7(c) shows how the SIMD unit executes these seven threads for re-converging at the immediate post-dominator. There are many empty slots in this figure and on average only 3.25 threads are enabled. It is also interesting to notice that the execution of the CFG of ~~Figure 10(c)~~Figure 3(c) is the same as Figure 7(c), which means AMD GPUs are also inefficient for this example.

~~Dynamic code expansion~~**Dynamic code expansion** occurs when different paths originating from a divergent branch pass through common basic blocks before the re-convergence point. For example, in Figure 7(c), time slots 7 to 11 are running dynamically expanded code because B3, B4 and B5 have been already executed in time slots 4, 5 and 6. This concept is defined against **static code expansion**, which inserts new instructions and increases the static binary size. During execution, dynamically expanded instructions will use the same PC values while statically expanded instructions will use different PC values.

The solution that reduces dynamic code expansion is to re-converge as early as possible. Figure 7(d) is an example where re-convergence happens much earlier than the immediate post-dominator. It saves execution time and has much better hardware resource occupancy. To achieve performance improvements as shown in Figure 7(d), the compiler should be capable of identifying the potential early re-converge points and inserting necessary check instructions. It also needs the support from hardware to efficiently compare the program counter (PC) of each thread to check for re-convergence. There is no commercial technology that can achieve the efficiency shown in this example and thus there is still a great deal of room for improvement in executing unstructured control flow in SIMD

processors.

The inefficiency of re-convergence at immediate post-dominators exacerbates the problem of branch divergence. If unstructured control flow can be handled more efficiently, some new language semantics, such as C++ try/catch style exceptions, can be added to ~~the~~ current programming model. Furthermore, compilers do not have to generate structured control flow as in Figure 10(c)~~,~~ if hardware more efficiently supports unstructured control flow.

2.4 Executing Arbitrary Control Flow on GPUs

Consequently, there are three ways to run programs with arbitrary control flow~~s~~ on different GPU platforms in an efficient (and hence portable) manner:

- The simplest method is to let compilers have the option to produce IR code only containing structured control flow~~s~~. This IR code then can be compiled into different back-ends. This method may miss some optimization opportunities, but it is simplest to implement.
- Use a JIT compiler to dynamically transform the unstructured control flow to structured control flow online when necessary, i.e., the target GPU does not support unstructured control flow. The dynamic compilation may introduce some inevitable overhead.
- The most promising method is to develop a new technology (with support from both compiler and hardware) to replace current approaches to fully utilize the early re-convergence opportunity that is illustrated in Figure 7(d).

This paper presents an approach to the second option above - transformation of the unstructured control flow to structured control flow in Section 3. The third option is remained as the future work.

3 Control Flow Transformations

The principal result of Zhang's (2004) work is that the repeated application of three primary transformations can provably convert all possible unstructured programs into a structured format. However,

their technique only applies to the programming language level instead of the IR level. To perform similar transformations at the PTX IR level some extra work is needed and the three original transformations have to be adapted because there is no simple one-to-one mapping between CFG and syntactic constructs. For example, syntactic constructs are flat but CFG is a two dimensional structure. The adapted transformations are conceptually and functionally equivalent to the ones used in Zhang's work (the detailed algorithm and correctness proof can be found in their original work) and can be explained through the application of three primitive transformations.

- Cut: The Cut transformation moves the outgoing edge of a loop to the outside of the loop. For example, the loop in Figure 8(a) has two unstructured outgoing edges, E1 and E2. What cut transformations do is i) using three flags to label the location of the loop exits (*flag1*, *flag2*, and *exit* in Figure 8(b)); ii) combining all exit edges to a single exit node ~~(Figure 8(c))~~; iii) using three conditional checks to find the correct code to execute after the loop(Figure 8(d)). It should be noted that after the transformation the CFG in this example is still unstructured and needs other transformations to make it structured.

- Backward Copy: Backward Copy moves the incoming edges of a loop to the outside of the loop. For instance, Figure 9(a) has an unstructured incoming edge E1 into the loop. To transform it, the backward copy uses the loop peeling technique to unravel the first iteration of the loop (Figure 9(b)) and points all incoming edges to the peeled part (Figure 9(c)). In this example, the CFG after the transformation is also unstructured. This transformation is rarely needed (see the experiment part in Section 4) because usually neither programmers nor compilers would create loops with multiple entries.

- Forward Copy: Forward Copy handles the unstructured control flow in the acyclic CFG. After Cut and Backward Copy transformations, there are no unstructured edges coming into or going out of loops. As a consequence, CFGs inside every loop can be handled individually and all structured loops can be collapsed into abstract single CFG nodes. Forward Copy eliminates all remaining unstructured branches by duplicating their target CFG nodes when traversing the CFG in the depth-first order. For example, in Figure 10(a), B5 needs to be duplicated because

edge E2: B4→B5 is unstructured (Figure 10(b)). Similarly, in Figure 10(b), subgraph {B3, B4, B5 and B5'} also has to be duplicated because edge E1: B2→B3 is unstructured (Figure 10(c)). If Forward Copy is performed multiple times, some subgraphs may be duplicated more than once and it may eventually lead to exponential code expansion. The final result shown in Figure 10(c) duplicates B5 three times and duplicates B4 and B3 once respectively. Actually, ~~Figure 10(c) spans all possible paths between the entry node and the exit node~~ every paths between the entry node and the exit node of Figure 10(c) is a possible execution path of Figure 10(a) .

Figure 11 compares the code duplicated by Forward Copy and the dynamically expanded code caused by re-convergence at the immediate post-dominator (Figure 11(b) and Figure 11(c) use the same color to represent the same basic block), it is interesting to see that they are exactly the same. Moreover, the execution of the re-convergence at the immediate post-dominator can be drawn as a tree (the red tree in Figure 11(c)), where each path of the tree stands for an execution path of the program. This red tree is also the same as the CFG of Figure 11(b). ~~In fact these two trees both traverse the original CFG in a depth first order and they are both the depth first spanning trees of the original CFG.~~ This is not a coincidence and can be generalized because they are both the depth first spanning trees of the original CFG. A simple proof is like this: if all threads in a warp follow different paths, which is the worst case of the execution, the warp would diverge in every CFG node which is the same as traversing the CFG in the depth-first order. Meanwhile, Forward Copy duplicates all target node of the unstructured edges in the depth-first order and finally no node (except the exit node) has two or more parent nodes (otherwise, it is still unstructured) and the transformed CFG becomes a binary tree. Hence, Forward Copy, in general, can be used to measure the worst case of dynamic code expansion of immediate post-dominator re-convergence without running the program.

All the above three transformations will cause static code expansion since they insert new instructions into the original program. In SISD processors, static code expansion has disadvantages of increased binary size and instruction cache footprint. This code expansions is even more problematic

for SIMD architectures because paths through duplicated blocks cannot be executed in lock-step by threads in a warp since these blocks use different PC values. Moreover, the~~The~~ cut transformation ~~,~~ ~~especially,~~ has to use several new variables to store flag values, which introduces new register pressure. It needs to use more conditional branches as well when exiting the loop, which may cause more divergence in the GPU architecture (see Section 2.2).

Only using the above three transformations is not sufficient since program structure must also be maintained - such as which basic blocks form a loop or the nesting level of a control block. A data structure called a *control tree* (Muchnick 1997) can provide this information, which basically describes the components of all control flow patterns and their nested structures. Figure 12(b) shows a CFG and its corresponding control tree. It should be ~~noticed~~noted that all unstructured control flow~~s~~, such as subgraph {B1, B2, B3} in Figure 12(b), are also included in the tree.

Including the control tree construction and the three primitive transformations, the process of transforming an unstructured CFG to a structured CFG has four steps:

1. Build a control tree and identify unstructured branches and some basic structured control flow patterns in the CFG including if-then, if-then-else, self-loop, for-loop, and do-while loop,

2. Collapse the detected structured control flow pattern into a single abstract node and update the control tree. The aim of this step is to simplify the search space of the following pattern matching steps.

3. If all the children nodes of an unstructured node are ~~all~~ structured, use the three primitive transformations to transform it into structured control flow. If the children nodes still contain unstructured control flow, wait until all children become structured. Otherwise, ~~it~~the transformation would introduce more unstructured parts into the CFG. For example, in Figure 10, edge E2 must be transformed before edge E1.

4. Update the control tree to reflect the transformation. If there is no unstructured part remaining, the process finishes. Otherwise, go back to the second step to transform the ~~rest~~remaining unstructured parts.

Figure 12 shows all steps of transforming an unstructured CFG into a structured CFG. It first builds a control tree and finds that subgraph B1, B2, B3 is unstructured (Figure 12(b)). Second, it collapses the self loop into a single node and updates the control tree (Figure 12(c)). Third, it uses Forward Copy to duplicate B3 (Figure 12(d)). Finally, it updates the control tree and determines that there are no more unstructured parts and the whole process is completed (Figure 12(e)). Figure 12(f) is the final result.

This transformation is not only useful in characterizing unstructured control flows as discussed above. It can also be used in dynamic compilers which are used in heterogeneous systems. Since the support for unstructured control flows in different GPU devices is different, this kind of transformation allows the program to run on several different back-ends which is very useful for large clusters comprised of different GPU backends.

4 Experimental Evaluation

This section evaluates how often unstructured control flows occurs in real GPU programs and how they may impact the performance over a large collection of CUDA benchmarks from the CUDA SDK 3.2 (NVIDIA 2010), Parboil 2.0 (Impact Research Group 2009), Rodinia 1.0 (Che et al. 2009), Optix SDK 2.1 (Parker et al. 2010) and some third party applications. The CUDA SDK contains a large collection of simple GPU programs. Parboil benchmarks are compute intensive. Rodinia's collection is chosen to represent different types of GPU parallel programs which are more complex than those in the CUDA SDK. Optix SDK includes several ray tracing applications. The third party GPU applications used include a 3D renderer called *Renderer* (Tsiodras), a radiative transport equation solver called *mcrad* (Fernando 2004) , and a monte carlo simulator called *mcx* (Fang and Boas 2009).

As for the compilation tools, NVCC 3.2 is used to compile CUDA programs to PTX code. Optix SDK benchmarks are running under Optix's own execution engine. A GPU compilation infrastructure, Ocelot 1.2.807 (Diamos et al. 2010), is used for several other purposes: back-end code generation, PTX transformation, functional emulation, trace generation and performance analysis.

4.1 Static Characterization

The first set of experiments attempts to characterize the existence of certain types of control flow in existing CUDA workloads by using the unstructured to structured transformations introduced in Section 3. The transformation is implemented as a static optimization pass in Ocelot, and it is applied to the PTX code of all benchmarks. The optimization can detect unstructured control flow and classify ~~them~~it by the type of transformations used (Cut, Backward Copy, or Forward Copy). The correctness of the transformation is verified by comparing the output results of the original program and the transformed program. Table 1 shows the number of applications having unstructured control flow in four examined GPU benchmark suites. Out of the 113 applications examined, 27 contain unstructured control flow, indicating that, at the very least an unstructured to structured compiler transformation is required to support general CUDA applications on all GPUs. It is also the case that more complex applications are more likely to include unstructured control flow. Almost half of the applications in the Rodinia and Optix benchmark suites include unstructured control flow.

Table 2 shows the usage of different transformations. The second column is the number of branch instructions in each benchmark. The third to the fifth columns show the number of times each transformation is used for every benchmark. The statistics show that Backward Copy appears to be nonexistent in current workloads which follows the common practice that programmers rarely write a loop with multiple entries. Cut transformations are necessary in programs that involve loops, but the shallow levels of nesting of GPU programs, especially those simple programs, makes this operation less common. Forward transformations are used most often. Further ~~research~~analysis shows that short-circuiting is the main trigger of these transformations. As explained in Section 2.2, short-circuiting does not run efficiently on the GPU platform.

The sixth and seventh column of Table 2 is the static code size of the benchmarks before and after the transformation. Static Code size is calculated by counting the PTX instructions of the benchmark. Usually the larger its code size is, the more complex control flow the program may have and the more transformations it needs. Benchmarks that have one Cut and zero Forward Copy, such as *path_trace* and *heightfield*, show that the number of instructions inserted by Cut is small. However, this is not

the case for the Forward Copy. In the benchmark *mummergpu*, its code size is almost doubled by 26 Forward Copy transformations. The code size increment caused by Forward Copy depends on the size of the shared CFG nodes that need to be duplicated. Column eight is the relative static code expansion. Figure 13 shows the code expansion of those benchmarks using at least one Forward Copy with an average of 18.68%. For those benchmarks having a large number of transformations, such as *mummergpu*, *mcx*, and *Renderer*, the static code expansion is significant.

Among all the benchmarks, *Renderer* has far more transformations than the rest. Other graphics benchmarks (*particles*, *Mandlebrot*, Optix SDK suite) also have more unstructured control flow on average. It is fair to say that graphics applications have great potential to improve performance if unstructured control flow could be handled more efficiently.

The ninth column lists the transformation time of each benchmark. The overhead is basically proportional to the number of taken transformations. Most of the benchmarks uses less than 1 millisecond to finish the transformation. Only *Renderer* takes more than 0.1 second because it has more than 8,000 unstructured branches. Considering the fact that the transformation is a one time investment and complex programs usually run for a long time (e.g. tested ray tracing programs can run forever), this overhead is affordable even if applied in runtime.

4.2 Dynamic Characterization

4.2.1 Impact of IPDOM

In the absence of support for re-convergence for unstructured control flow *vs.* immediate post-dominator, we use the functional emulator provided by Ocelot to count the number of instructions executed due to the lack of earlier re-convergence. For example, the instructions executed in the red circle of Figure 14. We perform measurements over code segments that meet three requirements: i) start with an unstructured branch; ii) the threads are divergent; iii) ends with the immediate post-dominator of the unstructured branch. The instructions satisfying above three requirements can be optimized by re-converging at the earlier point.

To find these instructions, we first determined the basic blocks (CFG nodes) between an unstruc-

tured branch and its immediate post-dominator by using our static transformation. These basic blocks might be dynamically expanded at runtime (like B3, B4, and B5 in Figure 14). Then, we used the emulator to count the number of times these basic blocks will run in a divergent *warp*. For example, in Figure 14, B3 and B4 each run~~s~~ twice in the divergent ~~warp~~*warp* and B5 runs four times. Assuming each basic block has 1 instruction for the simplicity of counting instruction number, we can say at most 8 instructions may miss the early re-convergence (actual number of dynamic expanded instructions is 5, executed in time slots 7–11). Although this method overestimates the performance degradation, the overestimated part is limited to the initial execution of these instructions. Tak~~e~~ing Figure 14 as an example, the overestimated part is time slots 4–6 during which B3, B4 and B5 are executed for the first time.

Table 3 shows the upper limit of dynamic code expansion for benchmarks using Forward Copy. Some benchmarks are not included ~~due to the issues of the emulator~~ because the emulator cannot correctly execute them yet. The results vary greatly. Four simple benchmarks of CUDA SDK have very low value because the unstructured part is executed infrequently or *warps* do not diverge when executing them. However, other benchmarks, such as *Renderer* and *mcx*, have a significant value meaning the application repeatedly executes these unstructured control flow~~s~~. In this case, not having earlier re-convergence will impact the performance significantly. It is also interesting to notice that benchmark *tpacf* has low static code expansion but high dynamic code expansion, which means the unstructured part is executed very frequently.

4.2.2 Performance Evaluation of the Transformation

In this experiment, we compare the performance of benchmarks listed in Table 3 before and after the nonstructural to structural transformation to explore its impact. In Figure 15, **IPDOM** uses IPDOM to re-converge those unstructured benchmarks, **STRUCT** first applies the proposed transformation and then use IPDOM to re-converge. The performance result shows that **STRUCT** is slightly worse than **IPDOM** due to the execution of static code expansion. The benchmarks such as *mergeSort* have no noticeable difference between two methods since they do not spent much time in unstructured code

as shown in Table 3 and do not need to apply lots of transformations. **IPDOM** works over 1% better than **STRUCT** for the benchmarks such as *Renderer* since they stayed in the unstructured part longer than the other benchmarks.

4.3 Case Study

In this experiment, we modified the Ocelot emulator and rewrote the *mummergpu* benchmark to force it to re-converge as early as possible. The benchmark rewriting includes adjusting the PTX code layout, so that re-convergence points appear at the earliest point and let the emulator to check the potential re-convergence point. We measured the dynamic instruction count (instructions dynamically executed by all threads) of two mechanisms: re-converging at immediate post-dominator and re-converging at the earliest point. The result shows that re-converging at earliest point can reduce **14.2%** (from 53616778 to 46008916) dynamic instructions, which further demonstrates that it is a promising research area.

5 Related Work

SIMD architectures have been designed with basic support for control flow since their large-scale deployment in the 1960s. Since that time, new designs have been incrementally augmented with compiler-assisted hardware support for non-nested and eventually all structured control flow and eventually all hammock graphs (Zhang and H. D'Hollander 2004,). These designs have culminated in support for all forms of unstructured control flow without any static code expansion since they allow arbitrary branches instead of being restricted to the nested structured control flow. However, to the best of our knowledge, all schemes employed in existing designs experience dynamic code expansion when executing unstructured control flow. A recently proposed technique, dynamic warp formation, introduced by Fung et al (Fung et al. 2007) potentially addresses this problem. However, it requires fully-associative comparisons across at least all warps on a multi-processor that cannot be feasibly implemented in modern power-limited designs.

ILLIAC IV (Bouknight et al. 1972), which is in general considered to be the first large-scale

SIMD supercomputer, was designed around the concept of a control processor that maintained a series of predicate bits, one for each SIMD lane. Its instruction set could be used to set the predicate bits to implement simple common structured control flow~~s~~ such as if-then-else blocks and loops.

The primary limitation of a single predicate register is its inability to handle programs with nested control flow. In 1982 the CHAP (Levinthal and Porter 1984) graphics processor introduced the concept of a stack of predicate registers to address this problem. CHAP includes explicit instructions for if, else, endif, do, while statements in the high level language. This is currently the most popular method of supporting control flow on SIMD processors and is also used by the AMD Evergreen and Intel GEN5 graphics processors.

To support unstructured control flow, a technique referred to as immediate post-dominator re-convergence was developed, which extends the concept of a predicate stack to support programs with arbitrary control flow (Fung et al. 2007). This is done by finding the immediate post-dominator for all potentially divergent branches and inserting an explicit re-converge instruction. During execution, predicate registers are pushed onto the stack on divergent branches and popped when re-convergence points are hit. In order to resume execution after all paths have reached the post-dominator, the program counter of the warp executing the branch is adjusted to the instruction immediately after the re-converge point.

All of the previous techniques have been implemented in commercial SIMD processors. Dynamic warp formation is a technique originally described by Fung et al. (2007) that increases the efficiency of immediate post-dominator re-convergence by migrating threads between warps if they are executing the same instruction. ~~However, the power and hardware complexity to support detecting this and dynamically creating a new warp from a pool of threads at the same PC, which requires fully associative PC comparisons across active warps every cycle and register file changes may outweigh the performance advantages.~~ This scheme takes advantage of the fact that most programs executed by GPUs are SPMD programs where all threads in all warps begin from the same program entry point. This makes it likely that threads from different warps will be at the same PC in the program at the same time. Fung suggests adding hardware support for detecting this case and dynamically creating

a new warp from a pool of threads at the same PC. In cases with where warps have many disabled threads, this can lead to significant performance improvements. However, the power and complexity of the hardware support required to perform fully associative comparisons across active warp PCs every cycle coupled with changes to the register file structure may potentially outweigh the performance advantages. Like post-dominator re-convergence, this scheme supports all program control flow. In the newly extension of the dynamic warp formation, Fung et al. proposes the concept of Likely-Convergence Points (Fung and Aamodt 2011), which is basically the probable earlier re-convergence points, without giving a practical approach to find these points. They found speedup when iterating Likely-Convergence Points with IPDOM and their dynamic warp formation framework.

Very recently, Diamos et al. (2011) first time systematically address the problem of unstructured control flow in SIMD processors by replacing IPDOM with a technique called *thread frontier*, which includes a static algorithm to find earlier re-convergence points and a hardware framework to let the program re-converge at these points. To get its best performance which is 1.5%-633% speedup over IPDOM, their method relies on a specific scheduling order which may not be possible to get statically and custom hardware support. Their findings prove the argument of this work that this is an area worth more attention and research investment.

As to the area of GPU application characterization, Kerr et al. (2009) and Goswami et al. (2010) respectively characterized a large number of GPU benchmarks by using a wide range of metrics covering control flow, data flow, parallelism, and memory behaviors. Goswami also researched the similarities between different benchmarks. Their studies are valuable for future GPU compilation and microarchitecture design. ~~Instead, our work focuses~~ Our work differs from theirs in that we target on the execution of unstructured control flow in GPUs and provides insight, characterizations, and suggestions for future GPU designs.

6 Conclusions

This work addresses the problem of running arbitrary programs on any GPU device. The current state of practice is not satisfactory since the support of unstructured control flow is very poor. Some

GPU devices do not support unstructured control flow at all, while others do not support it efficiently because they will miss the earliest re-converge point. We propose an IR level control flow transformation that can turn an unstructured control flow into a structured one. This transformation is used to characterize the existence of unstructured control flow in a large number of benchmarks. The result verifies the importance of the problem. Further, the transformation is also useful in a dynamic compiler used in heterogeneous systems.

In the future, we will focus on automatically finding earliest re-convergence points in an unstructured control flow graph by using different compiler and hardware techniques to improve execution efficiency of arbitrary programs on GPUs. Moreover, we are also considering a structured to unstructured transformation. The motivation of this reverse transformation is that if we have the technique to re-converge at the earliest point, the performance of an unstructured CFG is better than its functionally equivalent structured CFG (see Section 2.3). The process is simple: ~~which~~it continuously finds identical subgraphs and then merge them. ~~For example, if reverse transforming Figure **??**(a), the first step is to find and merge identical B5 nodes (Figure **??**(b)) and the second step is to find and merge identical subgraph {B3, B4, B5} (Figure **??**(c)).~~ The difficulty here is searching identical CFG subgraphs which needs to compare the register name and pointer value of all the instruction operands.

7 Acknowledgements

This research was supported by NSF under grants IIP-1032032, CCF-0905459, and OCI-0910735, by LogicBlox Corporation, and equipment grants from NVIDIA Corporation. We would also like to thank Andrew Kerr, Tri Pho and Naila Farooquie for helping us set up the experiments. Tips from the anonymous referees also greatly helped shape this paper.

8 Author Biographies

Haicheng Wu is a PhD student in the Computer Architecture and Systems Lab at the Georgia Institute of Technology, under the direction of Professor Sudhakar Yalamanchili. He received his B.S. in

Electrical Engineering from Shanghai Jiao Tong University in 2006 and his M.S. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2009. He used to focus on the dynamic thread management in the mainframe computers. Currently, he is moving and extending his prior research results to the heterogeneous architectures.

Gregory Diamos is a PhD student in the Computer Architecture and Systems Lab at the Georgia Institute of Technology, under the direction of Professor Sudhakar Yalamanchili. He received his B.S. and M.S. in Electrical Engineering from the Georgia Institute of Technology in 2006 and 2008, respectively, where he focused on architecture techniques for controlling PVT variations. His current research interests follow the industry shift from ILP to many core architectures, where mounting communication requirements place increasing demands on on-chip interconnection and the ability to tightly integrate heterogeneous architectures offers the potential for dramatic improvements in efficiency at the cost of increased design complexity; his research is directed toward maintaining this efficiency while reducing design complexity.

Jin Wang is pursuing her PhD under the direction of Professor Sudhaka Yalamanchili in Georgia Institute of Technology. She received her B.S. in Electrical Engineering from Shanghai Jiao Tong University in 2007 and her M.S. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2010. Her research interests are simulator and compiler support for heterogeneous computer architecture.

Si Li is a PhD student at Georgia Institute of Technology working in the area of computer architecture. His research interests involve the impact of on-chip interconnect on the unique memory demand and cache systems in the massively parallel environment of general purpose GPU architecture. His other interests include power and performance in the domain of computer architecture.

Sudhakar Yalamanchili earned his Ph.D degree in Electrical and Computer Engineering in 1984 from the University of Texas at Austin. Until 1989 he was a research scientist at Honeywells Systems and Research Center in Minneapolis. He joined the ECE faculty at Georgia Tech in 1989 where he is now a Joseph M. Pettit Professor of Computer Engineering. He is the author of VHDL Starters Guide, 2nd edition, Prentice Hall 2004, VHDL: From Simulation to Synthesis, Prentice Hall, 2000, and co-

author with J. Duato and L. Ni, of Interconnection Networks: An Engineering Approach, Morgan Kaufman, 2003. His current research foci lie in addressing the software challenges of heterogeneous architectures and solutions to power and thermal issues in many core architectures and data centers. Since 2003 he has been a Co-Director of the NSF Industry University Cooperative Research Center on Experimental Computer Systems at Georgia Tech. Dr. Yalamanchili continues to contribute professionally with service on editorial boards and conference program committees. His most recent service includes General Co-Chair of the 2010 IEEE/ACM International Symposium on Microarchitecture (MICRO) and Program Committees for the 2011 International Symposium on Networks on Chip (NOCS) and 2010 IEEE Micro: Top Picks from Computer Architecture Conferences.

References

AMD (2009). *Compute Abstraction Layer (CAL) Technology: Intermediate Language (IL)* (2.0 ed.). AMD Corporation.

AMD (2010, March). *Evergreen Family Instruction Set Architecture Instructions and Microcode*.

Bouknight, W., S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick (1972, Apr.). The Illiac IV system. *Proceedings of the IEEE 60*(4), 369 – 388.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009, October). Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, Volume 9, pp. 44–54.

Cooper, K. D., T. J. Harvey, and K. Kennedy (2001). A simple, fast dominance algorithm. Technical report.

Diamos, G., B. Ashbaugh, S. Maiyuran, H. Wu, A. Kerr, and S. Yalamanchili (2011). Simd reconvergence at thread frontiers. In *Proc. of the 44th Annual International Symposium on Microarchitecture*.

Diamos, G., A. Kerr, S. Yalamanchili, and N. Clark (2010). Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Proceedings of PACT '10*, pp. 353–364.

ACM.

Dominguez, R., D. Schaa, and D. Kaeli (2011). Caracal: Dynamic translation of runtime environments for gpus. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pp. 5–11. ACM.

Fang, Q. and D. Boas (2009). Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Optics express 17*(22), 20178–20190.

Fernando, R. (2004). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education.

Ferrante, J., K. Ottenstein, and J. Warren (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 9*(3), 319–349.

Fung, W. and T. Aamodt (2011). Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 25–36. IEEE.

Fung, W. W. L., I. Sham, G. Yuan, and T. M. Aamodt (2007). Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, pp. 407–420. IEEE Computer Society.

Goswami, N., R. Shankar, M. Joshi, and T. Li (2010). Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10. IEEE.

Impact Research Group (2009). Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.

Intel (2009, January). *Intel G35 Express Chipset Graphics Controller Programmers Reference Manual*.

Kerr, A., G. Diamos, and S. Yalamanchili (2009). A characterization and analysis of ptx kernels.

Lattner, C. and V. Adve (2004). LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pp. 75–86.

Levinthal, A. and T. Porter (1984). Chap - a SIMD graphics processor. *SIGGRAPH Comput. Graph. 18*(3), 77–82.

Matsuoka, S. (2008). The road to TSUBAME and beyond. *High Performance Computing on Vector Systems 2007*, 265–267.

Muchnick, S. (1997). *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers.

NICS (2010). `https://keeneland.gatech.edu/`.

NVIDIA (2009, October). *NVIDIA Compute PTX: Parallel Thread Execution* (2.1 ed.). Santa Clara, CA: NVIDIA Corporation.

NVIDIA (2010). `http://developer.nvidia.com/cuda-toolkit-32-downloads`.

Parker, S., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, et al. (2010). OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG) 29*(4), 1–13.

Rixner, S., W. Dally, U. Kapasi, B. Khailany, A. López-Lagunas, P. Mattson, and J. Owens (1998). A bandwidth-efficient architecture for media processing. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pp. 3–13. IEEE Computer Society Press.

Tsiodras, T. `http://users.softlab.ece.ntua.gr/~ttsiod/`.

Zhang, F. and E. H. D'Hollander (2004). Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 231–245.

Suite	Number of Benchmarks	Number of ~~Transformed~~ Benchmarks with Unstructured Control Flow
CUDA SDK	56	4
Parboil	12	3
Rodinia	20	9
Optix	25	11
Total	113	27

Table 1: Existence of unstructured control flow~~s~~ in different GPU benchmark suites

Benchmark	Branch Instruction	Cut	Forward Copy	Backward Copy	Old Code Size	New Code Size	Static Code Expansion (%)	Transformation Overhead (ms)
CUDA SDK								
mergeSort	160	0	4	0	1914	1946	1.67	0.085
particles	32	0	1	0	772	790	2.33	0.067
Mandelbrot	340	6	6	0	3470	4072	17.35	3.000
eigenValues	431	0	2	0	4459	4519	1.35	0.113
PARBOIL								
bfs	65	1	0	0	684	689	0.73	0.036
mri-fhd	163	1	0	0	1979	1984	0.25	0.193
tpacf	37	0	1	0	476	499	4.83	0.042
RODINIA								
heartwall	144	0	2	0	1683	1701	1.07	0.072
hotspot	19	1	0	0	237	242	2.11	0.038
particlefilter_naive	29	3	5	0	155	203	30.97	0.115
particlfilter_float	132	2	4	0	1524	1566	2.76	0.108
mummergpu	92	2	26	0	1112	2117	90.38	1.056
srad_v1	34	0	1	0	572	595	4.02	0.031
Myocyte	4452	2	55	0	54993	62800	14.20	7.677
Cell	74	1	0	0	507	512	0.99	0.076
PathFinder	9	1	0	0	136	141	3.68	0.024
OPTIX								
glass	157	0	7	0	4385	4892	11.56	0.412
julia	1634	14	22	0	14097	18191	29.04	4.509
mcmc_sampler	101	0	3	0	4225	4702	11.29	0.319
whirligig	143	0	8	0	4533	5303	16.99	5.663
whitted	173	0	6	0	5389	5841	8.39	0.334
zoneplate	297	0	3	0	3397	3400	0.09	0.073
collision	101	0	4	0	2585	2595	0.39	0.034
progressivePhotonMap	127	0	4	0	3905	3960	1.41	0.077
path_trace	29	1	0	0	1870	1875	0.27	0.028
heightfield	46	1	0	0	1761	1771	0.57	0.097
swimmingShark	51	1	0	0	1990	2000	0.50	0.067
mcrad	415	11	10	0	4552	5238	15.07	1.491
Renderer	7148	943	179	0	70176	111540	58.94	105.160
mcx	178	0	9	0	2957	5527	86.91	2.489

Table 2: Unstructured to structured transformation statistics

Benchmark	Dynamic Code Expansion Area (number of instructions)	Original Dynamic Instruction Count	Dynamic Code Expansion Area (%)
Mandelbrot	86690	40756133	0.21%
mergeSort	0	192036155	0.00%
particles	8	277126005	0.00%
eigenValues	7100	628718500	0.00%
heartwall	749028	121606107	0.61%
mummergpu	11947451	53616778	22.28%
tpacf	2082509458	11724288389	17.76%
Myocyte	205924	7893897	2.61%
Renderer	462485018	279729298	84.21%
mcx	13928549604	20820693588	66.90%

Table 3: Upper limit of dynamic code expansion

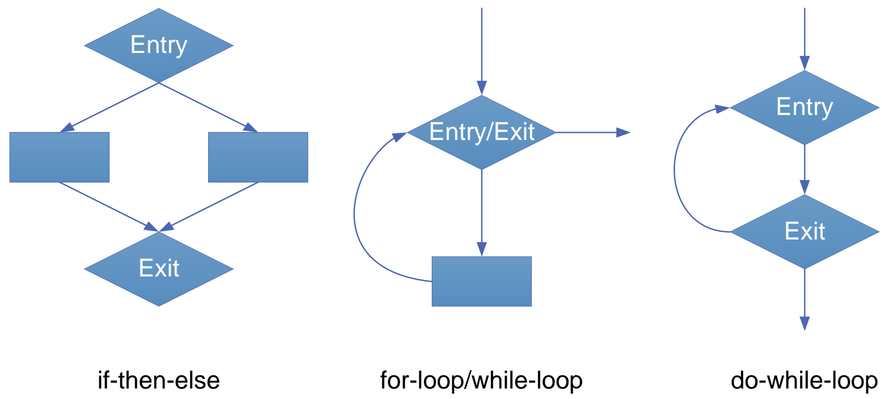

Figure 1: Examples of structured control flow: (a) if-then-else, (b) for-loop/while-loop, and (c) do-while-loop

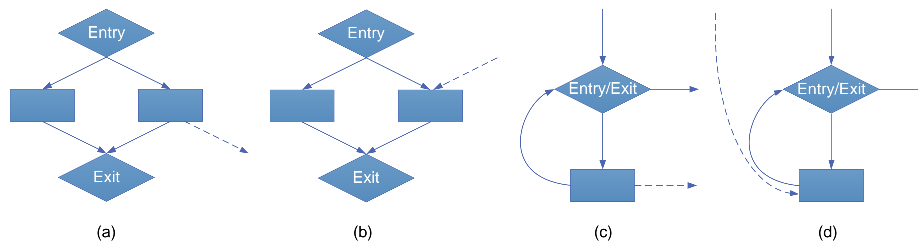

Figure 2: Examples of unstructured control flow: (a) if-then-else with extra outgoing edge, (b) if-then-else with extra incoming edge, (c) loop with extra outgoing edge, (d) loop with extra incoming edge

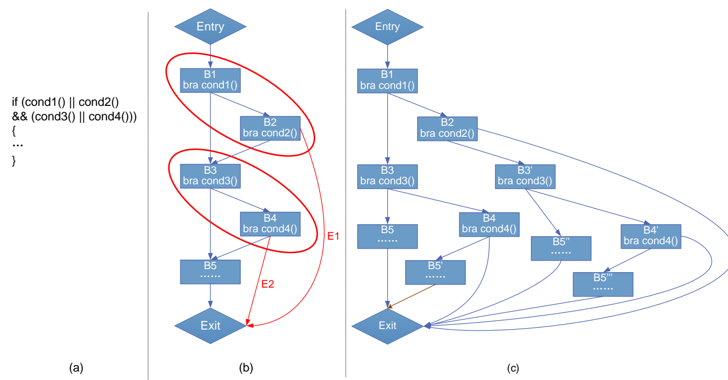

Figure 3: Example showing a compound condition that creates unstructured control flow: (a) code segment, ~~and~~ (b) CFG having unstructured control flow generated by short-circuit optimization, and (c) CFG used in AMD GPUs

.

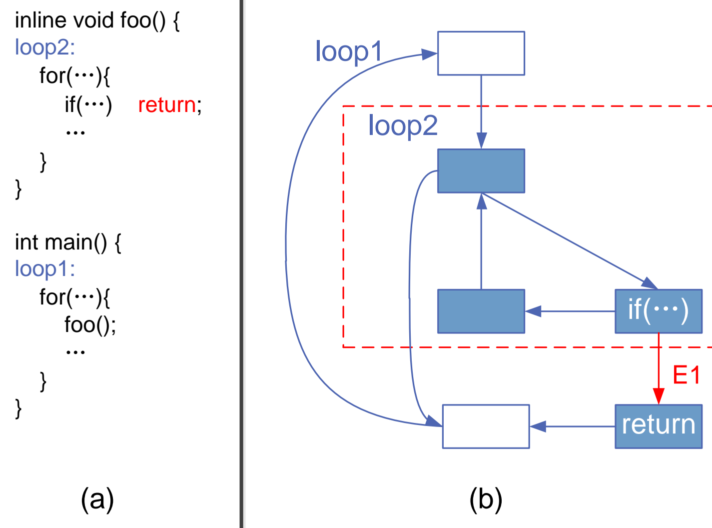

Figure 4: Example showing function inlining that creates unstructured control flow: (a) code segment and (b) CFG having unstructured control flow

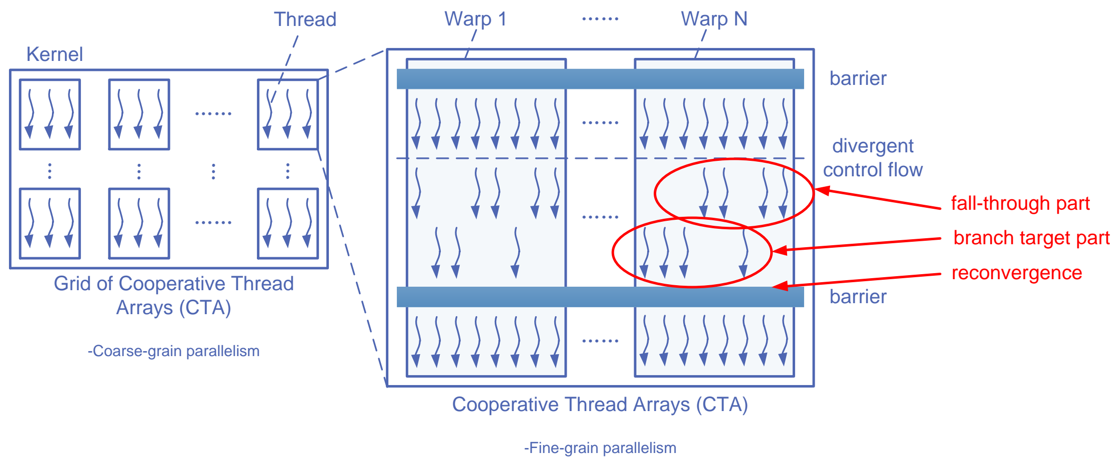

Figure 5: Execution Model of NVIDIA CUDA PTX

```
if (i < N)
{
      C[i] = A[i] + B[i]
}
```

```
ige r6, r4, r5
if_logicalz r6
uav_raw_load_id(0) r11, r10
uav_raw_load_id(0) r14, r13
iadd r17, r16, r8
uav_raw_store_id(0) r17,
r15
endif
```

(a) (b)

Figure 6: Example of AMD IL (a) C code, and (b) corresponding AMD IL

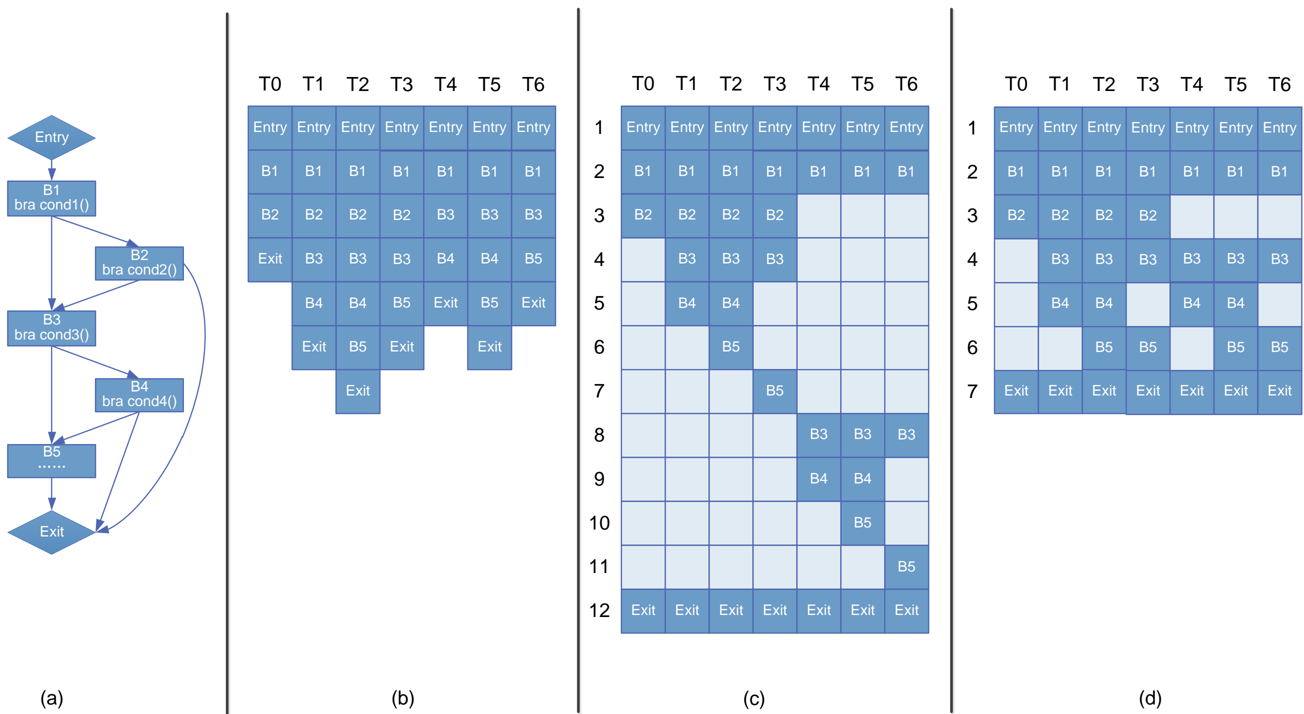

Figure 7: Example of mapping unstructured control flow into a SIMD unit: (a) unstructured CFG, (b) execution path, (c) re-converge at immediate post-dominator, and (d) re-converge at the earliest point

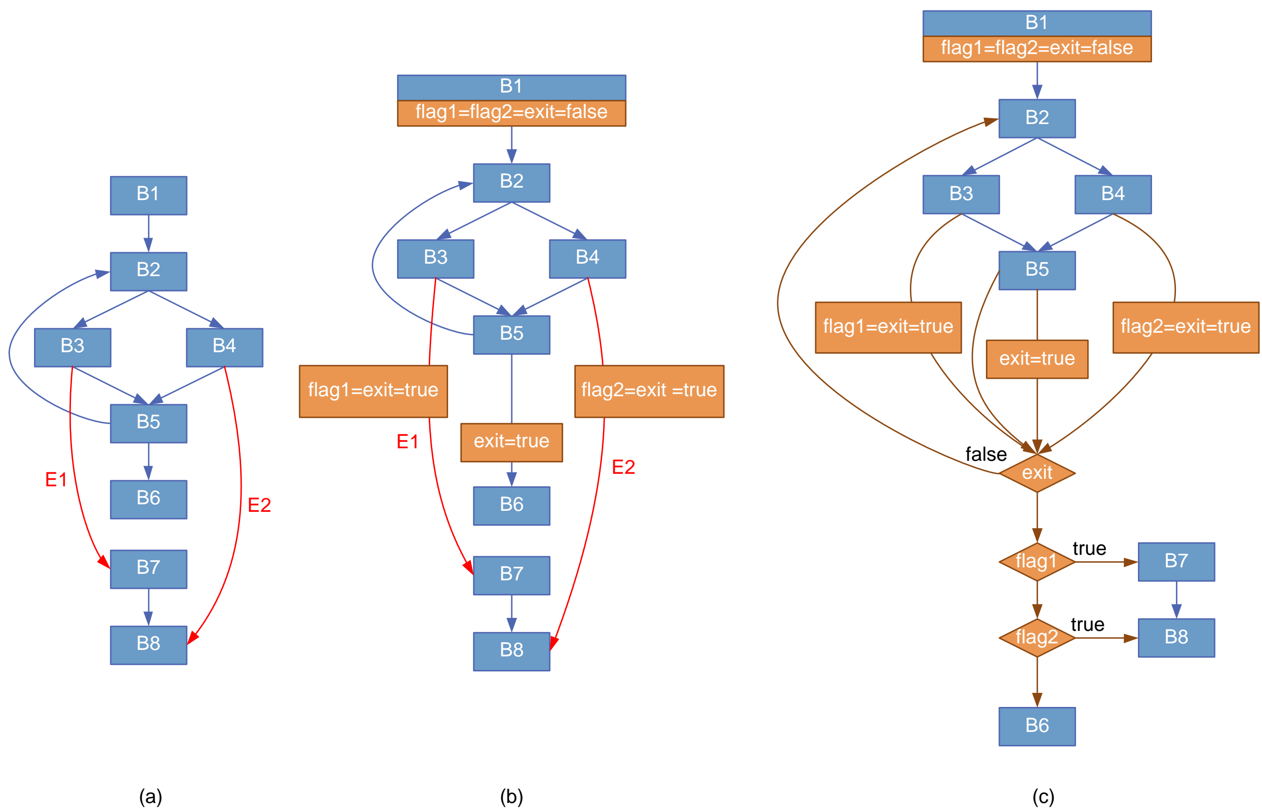

Figure 8: Example of a Cut transformation: (a) unstructured CFG (b) three flags are used to denote the loop exit location (*flag1*: exit from B3 or not; *flag2*: exit from B4 or not; *exit*: loop terminates or not) (c) combine all exit edges to a single exit node~~; (d)~~ and use three conditional checks to find the correct code to execute after the loop

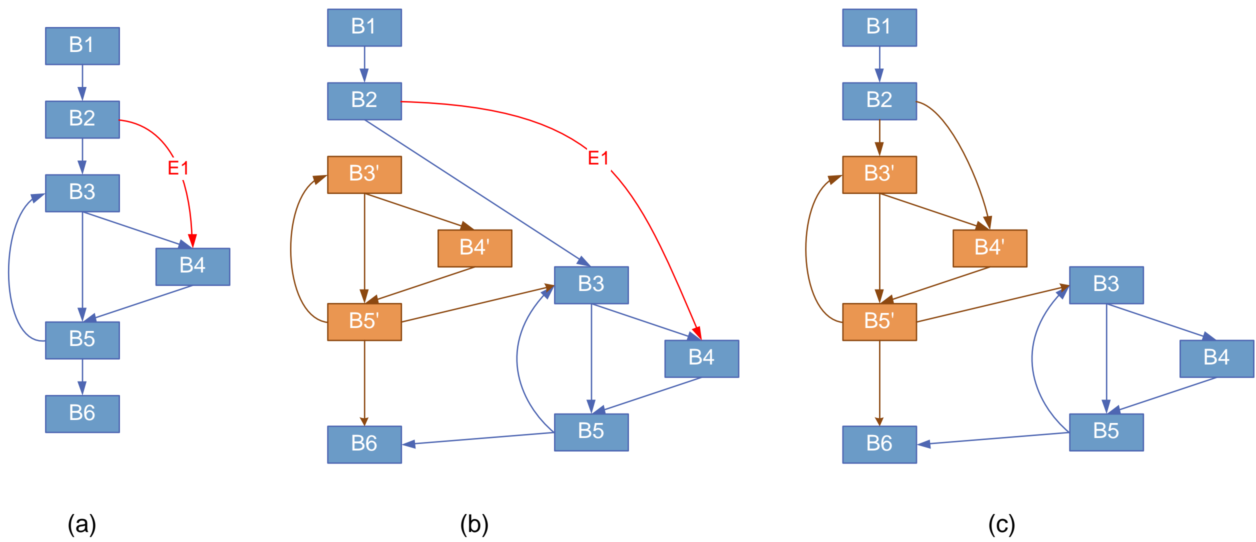

Figure 9: Example of a Backward Copy transformation: (a) unstructured CFG; (b) use loop peeling to unravel the first iteration; (c) point all incoming edges to the peeled part

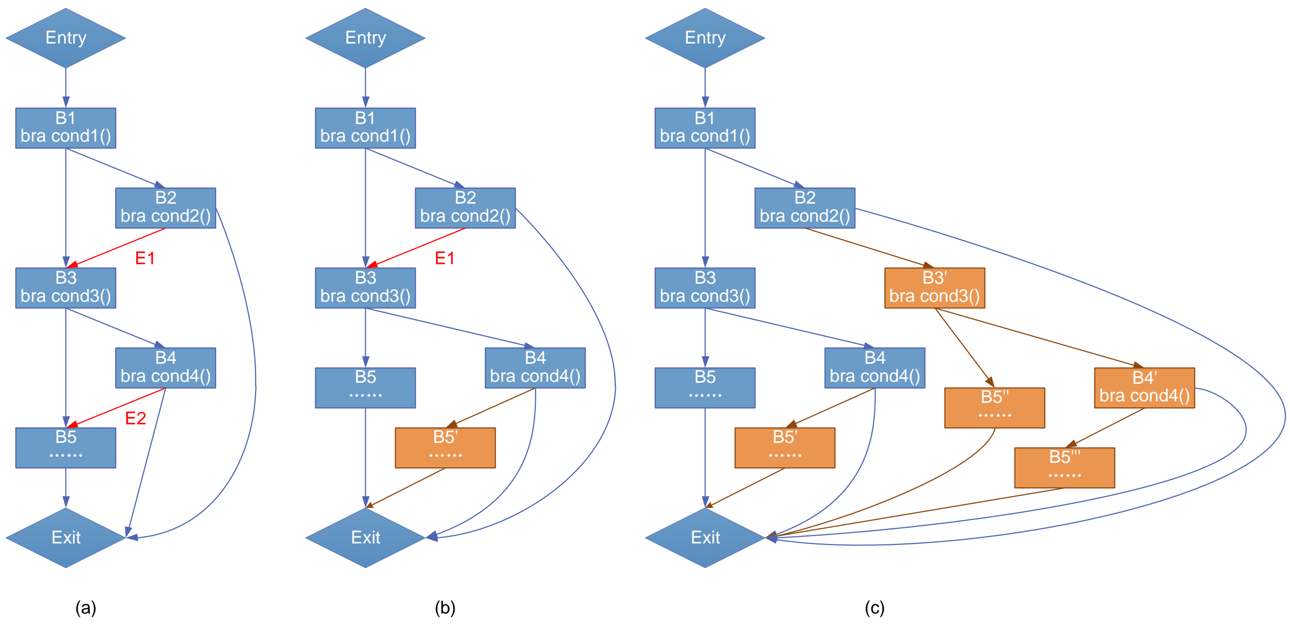

Figure 10: Example of a Forward Copy transformation: (a) unstructured CFG; (b) CFG after duplicating B5; (c) structured CFG

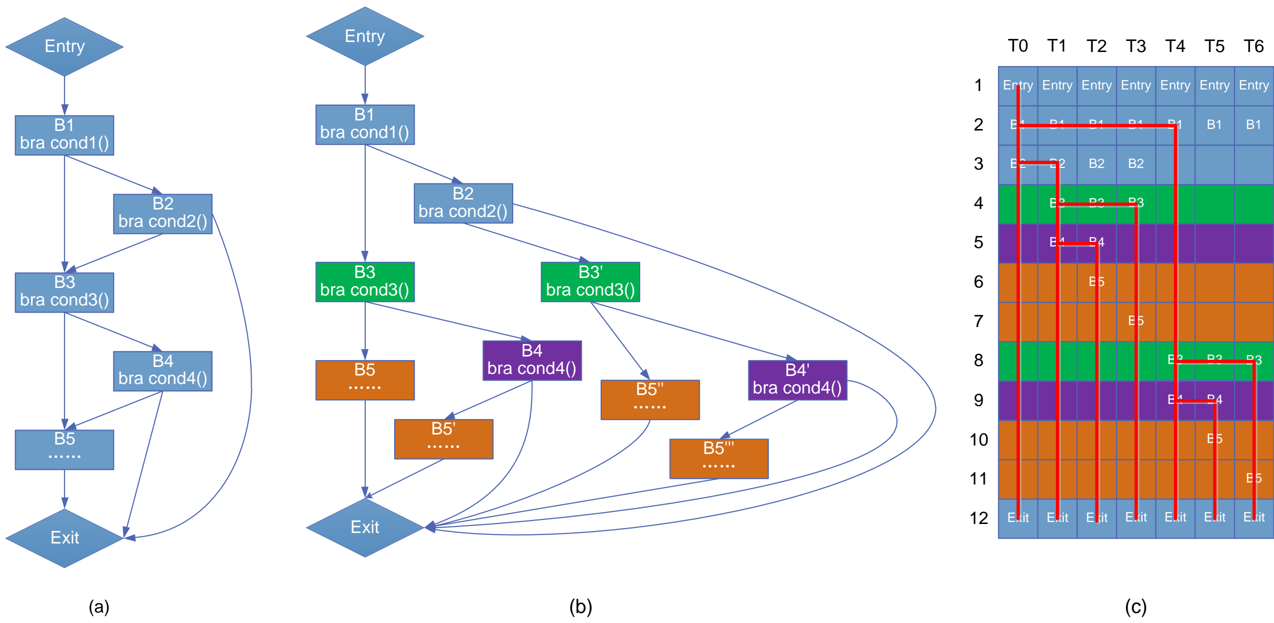

Figure 11: Relationship between Forward Copy and Re-convergence at the immediate post-dominator: (a) unstructured CFG; (b) result of Forward Copy / depth first spanning tree; (c) re-convergence at the immediate post-dominator

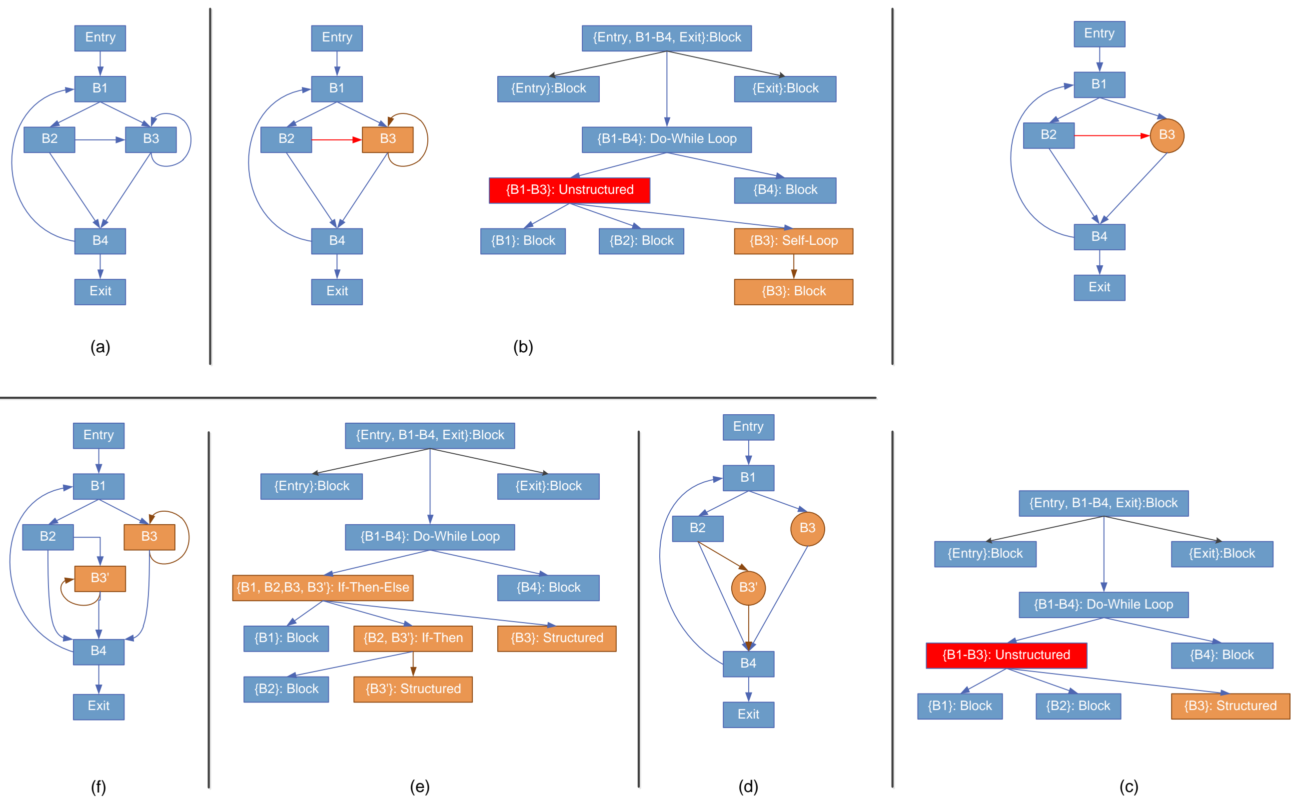

Figure 12: A Complete example of unstructured to structured transformation (a) original CFG; (b) build the control tree; (c) collapses the structured node and update the control tree; (d) Forward Copy transformation; (e) update the control tree; (f) final result

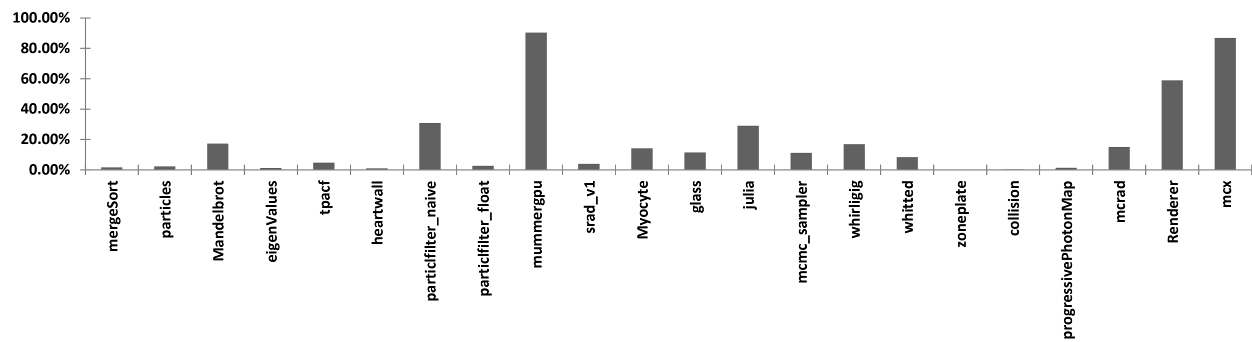

Figure 13: Static code expansion of benchmarks using Forward Copy

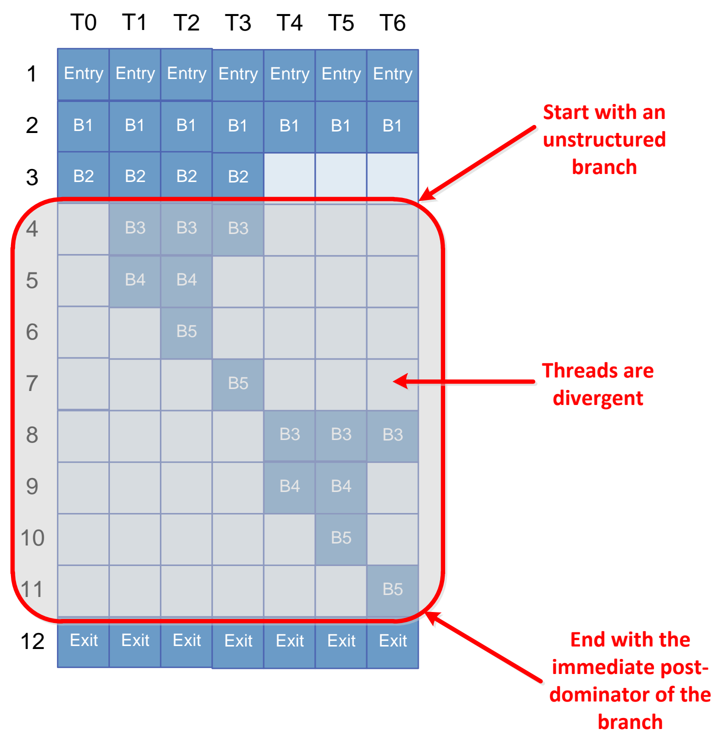

Figure 14: Example of measured dynamic code expansion statistics

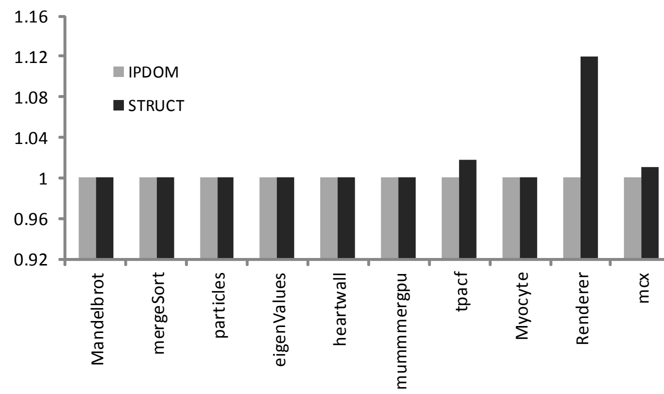

Figure 15: Dynamic Instruction Count before and after the Transformation (normalized to **IPDOM**)