Visualizing Potential Parallelism in Sequential Programs

Graham D. Price, John Giacomoni, Manish Vachharajani
University of Colorado at Boulder
{graham.price, john.giacomoni, manishv}@colorado.edu

ABSTRACT

This paper presents ParaMeter, an interactive program analysis and visualization system for large traces. Using ParaMeter, a software developer can locate and analyze regions of code that may yield to parallelization efforts and to possibly extract performance from multicore hardware. The key contributions in the paper are (1) a method to use interactive visualization of traces to find and exploit parallelism, (2) interactive-speed visualization of large-scale trace dependencies, (3) interactive-speed visualization of code interactions, and (4) a BDD variable ordering for BDD-compressed traces that results in fast visualization, fast analysis, and good compression. ParaMeter’s effectiveness is demonstrated by finding and exploiting parallelism in 175.vpr. Measurements of ParaMeter’s visualization algorithms show that they are up to seventy-five thousand times faster than prior approaches.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques; D.1.3 [Software]: Programming Techniques—Concurrent Programming

General Terms
Performance

Keywords
parallel programming, thread extraction, visualization

1. INTRODUCTION

Improving the performance of sequential programs not obviously amenable to parallelization is a longstanding concern. Until recently, software engineers could gain performance by waiting for processor core performance to improve. However, uniprocessor design challenges have forced manufacturers to improve performance with multicore designs. This shift forces software engineers to try and improve the performance of their applications with high-level redesigns focused on exposing parallelism.

Unfortunately, parallelizing sequential applications has proved difficult. Limit studies during the 1990s showed that sequential applications have considerable potential parallelism [2, 10, 13, 19] that is amenable to thread-level parallelism (TLP) [8] but is not accessible via hardware instruction-level parallel techniques (ILP) [19]. Unfortunately, automatic parallelization approaches have had limited scope [1] and not been able to exploit this TLP, though recent advances [12, 15] show promise for certain applications. Thus, converting the potential parallelism in these limit studies into real TLP remains elusive.

The parallelism observed in the limit studies is hidden inside large dynamic traces (e.g., billions of instructions). Thus, identifying the static code regions responsible for the observed parallelism and then extracting TLP from these regions requires tools that can locate candidate code sequences, analyze the dynamic data dependencies of these sequences, understand their interactions with other parts of the program, and relate the dynamic information to static analyses to determine how to proceed with parallelization. The key problem with this approach is that many of the interesting analyses and visualizations of the trace dependence structure are prohibitively expensive given commonly used large-trace compression and storage techniques.

Subsequent work addresses trace representation for analysis [11, 14], but not interactive visualization. Today, it is not clear (1) how to use these advanced representations for interactive visualization of potential parallelism, and (2) how to quickly visualize the results of analyses in a way that relates it back to the visualized parallelism. This paper presents ParaMeter, a visualization tool that helps find parallelism in large dynamic traces of sequential applications. The key contributions of this paper are:

1. A case study that uses interactive visualization of large traces to identify and extract potential TLP.
2. An algorithm for interactive visualization of potential parallelism in BDD-compressed traces [14].
3. An algorithm for interactive visualization of long-distance interactions between code sequences in a large trace.
4. A BDD variable ordering for BDD-compressed traces that yields good compression, rapid visualization, and fast analysis.

Results show that ParaMeter performs visualization and analysis scalably and is up to seventy-five thousand times faster than prior approaches, fast enough for interactive use.

The rest of this paper is organized as follows. Section 2 reviews material on Dynamic Instruction Number vs. Ready-time (DINxRDY) plots. Section 3 presents a case-study that uses ParaMeter to find parallelism in 175.vpr. Section 4 reviews related work.
2. BACKGROUND: DINxRDY PLOTS

Though parallelism found in prior studies [2, 10, 19] is not accessible to ILP techniques [19], it may yield to thread-level parallel (TLP) techniques. The Dynamic Instruction Number vs. Ready-time (DINxRDY) plot (originally introduced by Postiff et al. [13]) can be useful in identifying the potential TLP inherent in many sequential applications.

DINxRDY plots graphically represent parallel structures and expose potential threads for TLP [8]. Consider the hypothetical 5 instruction trace shown in Figure 1(a). The vertical axis represents the Dynamic Instruction Number (DIN) and the horizontal axis represents the earliest time at which an instruction can be scheduled (ready-time, RDY). For example, Figure 1(a) shows that the 3rd instruction in the trace (dynamic instruction 3, or DIN 3) was issued in cycle 3. Figure 1(b) shows the same trace under an ideal schedule (i.e., one cycle per instruction, perfect branch prediction, infinite hardware resources, etc.) that respects all data dependencies. This plot shows that DIN 3 is dependent on DIN 2 which in turn is dependent on DIN 1. Further, the plot shows that DIN 4 is not dependent on DINs 1, 2, or 3 because it is scheduled in the first cycle. Dependence analysis is needed to decide whether DIN 5 is dependent on DIN 1 or 4.

Iyer et al. observed that lines running from lower-left to upper right in DINxRDY plots form dependence chains of relatively nearby instructions in a program [8]. Further, diagonal lines that have overlapping x-extents suggest regions of code that might be convertible to TLP. Figure 2 shows a DINxRDY plot for 254.gap with groups of divergent dependence chains (DDCs) (circled in the figure) suitable for TLP extraction analysis. Figure 3 shows a plot for 175.vpr with three interesting DDCs labeled α, β, and γ.

Section 3 describes ParaMeter’s design and key algorithms. Section 6 evaluates ParaMeter’s performance. Section 7 concludes.

3. CASE STUDY WITH 175.VPR

This section shows how ParaMeter can help extract parallelism from sequential programs. In this case study, ParaMeter’s interactive visualization and analysis play a key role in (1) quickly identifying an opportunity for data parallelism, (2) recognizing the same as a candidate for pipeline parallelism, and (3) quickly exploiting the data-parallelism, all without prior knowledge of the application’s structure.

The case study proceeds as follows. First, the tool is used to explore and find divergent dependence chains (DDCs) in a trace of the 175.vpr SPEC INT benchmark (Section 3.1). Second, results from reverse slice analysis (a form of dependence analysis) are visualized to confirm the independence of the candidate DDCs (Section 3.2). Third, the confirmed DDCs are mapped back to the source code, helping the programmer verify and extract the TLP as data parallelism, and potentially pipeline parallelism (Section 3.3).

3.1 Finding Potential Parallelism

Figure 3 shows a ParaMeter DINxRDY plot region for a run of 175.vpr on its training input. Most notable in the plot are the three DDCs labeled α, β, and γ. From Iyer’s work [8], we know that these suggest the presence of either data parallelism or pipeline parallelism.

3.2 The Reverse Slice

While DDCs are likely candidates for TLP, there can be distant interdependencies limiting exploitable parallelism. Parameter provides interactive trace slice analysis to track dependencies within and amongst DDCs. Thus, a programmer can quickly categorize DDCs into potentially parallelizable and not-parallelizable.

Using ParaMeter’s slice-analysis visualization, we can discover that α, β, and γ likely represent real opportunities for parallelism. To see this, we request reverse-slice analysis for DDC γ (selected in Figure 4(a)) with results shown in Figure 4(b). Notice, γ (shown in orange) has dependencies, shown in red, on the distant α but has no dependencies on β. Similarly, β is dependent on α but not γ (Figure 4(c)).

Figure 1: Basic DINxRDY Plot with 5 instructions.

Figure 2: Dynamic instruction number vs. ready-time plot of SPEC CINT 2000 benchmark 254.gap. Circled areas represent potential threads.

Figure 3: Overview DINxRDY Plot of 175.vpr
Thus, β and γ could be data parallel threads because the regions: 1) are DDCs and, 2) have no interdependencies. Further, from the structure of the three regions we see that, even though the area of interest spans approximately 100 million dynamic instructions, there is opportunity for pipeline parallelism. Observe that though β and γ depend on α, β and γ still have x-extents that overlap α. This implies that early parts of α produce values that are read by early parts of β and γ, and later parts of α produce values that are read only by later parts of β and γ. Thus we may be able to build a 3-stage pipeline with an α-stage, β-stage, and γ-stage, where α-stage feeds data to β-stage and γ-stage, and where β-stage and γ-stage are running in parallel. Since these results are based on dynamic analysis, we must map DDCs back to the original source code for inspection and final parallelization.

3.3 Back to the Source

To help confirm that DDCs have no dependencies not captured in the particular trace analyzed, ParaMeter uses debugging symbols provided in the binary to map the interesting regions of the DINxRDY plot to the original source code.

Using ParaMeter’s plot-to-source mapping we find that α, β, and γ map to a set of memory allocation routines found in read_netlist.c (line 197) and two loops in “check_netlist” from read_netlist.c (line 798) respectively. Inspection reveals that the β and γ loops are data-parallel. Note that the code for α is not near the code for β and γ and locating it without ParaMeter requires extensive knowledge 175.vpr’s code.

Based on this analysis we parallelized β and γ. Pipeline parallelism between α, β, and γ could be exploited by leveraging techniques such as DSWP [12, 15] and the low-overhead point-to-point communication of FastForward [6]. Using ParaMeter we have also been able to quickly confirm other known opportunities for parallelism such as the commonly exploited TLP opportunity in 176.gcc [18, 23].

From this case study, notice how ParaMeter’s interactive visualization and analysis allowed us to quickly find and exploit opportunities for parallelism, including opportunities that spanned multiple methods, a non-trivial task.

4. BACKGROUND: COMPRESSED TRACES

Managing large dynamic traces is the key problem when performing whole program analyses. Consider that 1 billion 64bit values need ≈ 7.5 GB of uncompressed storage and that traces usually contain billions of instructions (10-1000s of GB). Therefore, trace analysis tool operate on compressed data. This section briefly reviews compression techniques optimized for sequential access and the more recent techniques, optimized for analysis, and used by ParaMeter for both analysis and its novel visualization algorithms.

4.1 Sequential Compressed Traces

Modern trace compression techniques [5, 22] optimize for sequential whole program analyses by storing data in a sequential no-random-access format.

Burtscher et al. [5] describe a predictor-based strategy requiring detailed information for only mispredictions. Thus, the resulting stream can be further compressed 10x or more by a standard stream compressor such as bzip2. Unfortunately, generating DINxRDY plots from such stream compressed data is too slow for interactivity as the decompressed data must streamed for each frame plot, which takes ≈ 5 minutes on a 2.0 GHz Pentium 4 for a trace containing only 500 million instructions. Worse, selecting and performing slice analysis on instructions requires two more passes through the complete data, resulting in 10 minutes to visualize the analysis.

Zhang et al. [22] address this problem by generating Reduced Ordered Binary Decision Diagrams (ROBDDs, or BDDs) to keep intermediate analysis results in a compact form in RAM. This makes it possible to analyze long traces, however, the analysis must still stream over the whole trace file, even when analyzing only a subset of the instructions. Thus analysis and visualization is prohibitively expensive.

Larus describes a SEQUITUR-based compression scheme for traces [11] which makes finding subsequences of instructions in the trace fast. However, visualization and analysis is still time consuming as we must stream over the entire region of interest explicitly, decompressing it on the fly.

4.2 BDD Compressed Traces

An alternative is to use a BDD-based trace representation as described by Price et al. [14], who based their work on that of Zhang et al. [22]. The representation permits rapid random access on the compressed form of the entire dynamic trace. We review the BDD-compressed format by summarizing the process by which traces are encoded as boolean functions, how the resulting functions are encoded in a BDD, and the importance of variable order for good compression. The examples are borrowed, with permission, from Price and Vachharajani’s paper [14]. Section 5 describes how the novel algorithms in this paper operate on BDD-compressed traces.

4.2.1 Traces as Boolean Functions

BDDs represent boolean functions, so storing traces, data dependence graphs, and the DINxRDY plot data in a BDD requires that they be converted to a boolean function first. The key is that boolean functions can encode arbitrary binary data. For example, if
the represented universe, \( \Omega \), consists of 4 elements \( \Omega = \{a, b, c, d\} \), then a 2-bit encoding can be used to represent each element, \( \{a \rightarrow 00, b \rightarrow 01, c \rightarrow 10, d \rightarrow 11\} \). With this encoding, we can create a boolean indicator function (i.e., characteristic function) that evaluates to true for any subset of the \( \Omega \). For example, the indicator function for the set \( \{a, b\} \) is \( I_{\{a,b\}} = x' \) where \( x \) is the variable for the most-significant bit (MSb) of the set encoding and \( x' \) is read as not \( x \).

Extending this notion, we can encode the different data sets necessary for our program analyses. Traces are just sequences of data and can be encoded by a set of tuples \((DIN, data)\) where the DIN is the dynamic instruction number. Therefore, instruction traces can be encoded as \((DIN, PC)\), where the PC is the program counter value. For example, the first and second instruction traces could be encoded as \((0, 0x100000f4)\) and \((1, 0x100000fe)\) corresponding to a trace starting at instruction address 0x100000f4 and continuing to 0x100000fe. To encode a trace with 64-bit program counter values (i.e., set of \((DIN, PC)\) tuples) as a boolean function we use the standard indicator function encoding described above, but this time for the Cartesian-product set \( \Omega = \mathbb{N}_d \times \mathbb{N}_p \) where \( \mathbb{N}_i \) is the set of integers from from 0 to \( 2^i - 1 \). We assume any DIN, \( d \), is in \( \mathbb{N}_d \) and any 64-bit PC value, \( p \), in \( \mathbb{N}_p \). Similarly, more complex data relationships can be encoded by simply joining the binary representations of arbitrary tuples into a single equation.

### 4.2.2 Boolean Functions as BDDs

BDDs can be viewed as compressed versions of binary decision trees. Figure 5(a) shows a binary tree for the three variable function \( f(x, y, z) = x' y + xy' + z \). For example, traversing the left edges of the graph we evaluate \( f(0, 0, 0) \) as 0. Reduced Ordered BDDs (ROBDDs) \[4\] are graph data structure in which each node corresponds to a boolean function (just as each node in a binary decision tree does). Converting a decision tree to an ROBDD (BDD henceforth) is covered in the literature \[4, 14, 17\]. Figure 5(b) shows the BDD for \( f \) under the variable ordering \((x, y, z)\). To compute \( f(0, 0, 0) \) with the BDD, we traverse the false arc of the X node, the false arc of the rightmost Y node and the inverting false arc of the Z node. Because we reached the 1 node through an odd number of inverting arcs, we find \( f(0, 0, 0) = 0 \) as before.

### 4.2.3 BDD Variable Ordering

For ROBDDs, variable ordering is the main factor in BDD size next to the actual data in the BDD. Figure 5(c) improves upon the compression achieved in Figure 5(b) by changing the variable ordering to be \((z, y, x)\). In practice, choosing a good variable ordering is critical as the resulting BDD size can differ by orders of magnitude. The literature contains a vast discussion of BDD variable ordering \[7, 9, 16\]. Section 5 discusses the ParaMeter’s variable ordering for good compression, rapid visualization, and fast analysis.

### 5. PARAMETER

This section describes ParaMeter’s novel interactive large-trace visualization algorithms. First, the section describes the insight behind ParaMeter’s high-performance DINxRDY visualization engine: quad-tree-based rendering. Second, it describes a variable ordering for BDD-compressed traces that gives good compression, rapid analysis, and allows the BDD to act as a compressed quad-tree. Third, it describes the algorithm used to generate the DINxRDY plots from the BDD trace representation. Finally, it describes novel algorithms for trace dependence-slice visualization.

#### 5.1 Quad-tree Rendering

The key insight behind all of ParaMeter’s rapid visualization algorithms is that a BDD representation, under the right variable ordering, can be treated as a quad-tree.

In graphics, a quad-tree is a hierarchical region-based decomposition of a two-dimensional space. Figure 7 shows a quad tree for the region decomposition in Figure 6. In the Figure, a node \( Ni \) corresponds to the region \( i \) in the figure. The root of the tree (N0) corresponds to the entire area to be plotted. The four children of a node correspond to four equally sized areas that cover the node’s region. For the root these are, from left to right, top to bottom in Figure 6, N1, N2, N3, and N4 (labels for 2-4 have been omitted for clarity). If a region has no points to be plotted (i.e., it is empty), the
5.2 Variable Ordering

Variable ordering is the critical element for the performance of BDDs in terms of both compression, analysis, and visualization. In this section we present ParaMeter’s variable order and explain how it allows ParaMeter to treat the BDD as a quad-tree. Section 5.6 empirically show that the ordering also gives to good compression and analysis speed.

In ParaMeter’s straight-forward trace representation, each BDD has 128 variables composed of two 64-bit integers representing each element of a tuple. In the discussion below, let $y = y_0 \ldots y_{63}$ represent the boolean variables (each bit) for the first element of each tuple, namely the DIN. Let $x = x_0 \ldots x_{63}$ represent the variables for the second half of the tuple which is either another DIN, the PC, or the ready-time. Further, let $y_{63}$ and $x_{63}$ represent the most-significant bits (MSBs) and $y_0$, $x_0$ the least significant bits (LSBs).

ParaMeter uses a simple static variable ordering to achieve its compressed quad-tree BDD representation. The ordering interleave the two 64-bit variables such that $y_i$ is adjacent to $x_i$ in order, and each $y_i$, $x_i$ pair is ordered from the MSb at the top of the BDD to the LSb at the bottom. Below we argue that this interleaving will yield both good compression and visualization performance on trace data, experimental confirmation is provided in Section 6.2.

Visualization.

ParaMeter’s interleaved ordering solves the visualization problem because the ordering induces BDDs that are essentially compressed quad-tree representations of the tuples. Consider the first 4 levels of the hypothetical BDD shown in Figure 8. For the purposes of illustration, we have only 3 DIN ($y$) and 3 RDY ($x$) variables in the BDD. Figure 9(a) shows the corresponding DINxRDY plot for this BDD. Notice how the false sub-graph (the sub-graph reached through the false arc) of the root node corresponds to the bottom half of the 1st cut shown in Figure 9(b), and the true sub-graph the top half. The 4 sub-graphs in Figure 9(c) correspond to the sub-graphs pointed to by level 2 in the figure. Note that there are only 2 nodes in level 3 because the upper-left and bottom-right trees are empty, and thus the appropriate edges from level 2 lead directly to false $^2$ (this will always be the case due to the canonicity of ROBDDs).

When plotting the DINxRDY plot, ParaMeter’s visualization speed is attributable in part to this early termination of the BDD search for empty regions. Recall that the DINxRDY plot is quite sparse so there are large empty regions in the BDD “quad-tree” (see Figure 6). In addition to the empty nodes, once a BDD sub-graph corresponds to a screen region that is entirely contained within a screen pixel, rendering can also stop, which also accelerates DINxRDY plotting when zoomed out, avoiding traversal of the bulk of the BDD. This allows our visualization algorithm to scale with screen resolution. In the next section, we discuss in detail, ParaMeter’s rendering algorithm.

Compression.

To conserve space, we omit a detailed explanation of why ParaMeter’s variable ordering results in good compression and instead rely on the empirical results in Section 6 to show that the ordering does indeed result in good compression. For those familiar with BDD variable orderings, ParaMeter’s variable ordering achieves good compression because:

1. Many tuples share the same high-order bits in the DIN. Thus, by placing the more significant $d_i$s at the top of the BDD, the representation of many tuples can share the BDD nodes for the least significant bits of their DIN.

$^2$Recall that in an ROBDD, there is only a constant true node, and the function has value false when this node is reached through an odd number of inverting arcs.

Figure 8: Simple BDD for quad-tree partitioning example in Figure 9.

Figure 9: Simple BDD/quad-tree partitioning example.
function plotDINxRDY_din(BDD_Node node, region_t region, parity_t parity)
if (node.isConstant())
    if (parity == even)
        region.fill_region(); // even parity, region has something, plot
    return
if(region.containedInPixel())
    return (left,right) = region.divideByDIN()
    plotDINxRDY_rdy(node.getTrueNode(), right, parity)
    plotDINxRDY_din(node.getFalseNode(), (bottom,top) = region.divideByDIN()
        node.falseArcIsInverting()?)
        region.getContainingPixel().plot()
        parity.flip();parity)

function plotDINxRDY_rdy(BDD_Node node, region_t region, parity_t parity)
if (node.isConstant())
    if (parity == even)
        region.fill_region(); // even parity, region has something, plot
    return
if(region.containedInPixel())
    region.getContainingPixel().plot()
return (region.containedInPixel())
    if (parity == even)
        region.fill_region(); // even parity, region has something, plot
    return
if(node.isConstant())
    if (parity == even)
        region.fill_region(); // even parity, region has something, plot
    return
    return node.falseArcIsInverting()?
        region.getContainingPixel().plot()
        parity.flip();parity)

Figure 10: Pseudo-code: DINxRDY plotting code.

2. In many of the compressed trace tuple sets (i.e., (DIN, RDY), (DIN,DIN), etc.) the value paired with a particular DIN shares the high-order bits in common with the DIN. By placing each such $d_i$ and $y_i$ near each other in the ordering (in our case adjacent to each other), we increase the likelihood that different tuples share a function in their corresponding cofactor expansion, which means they will share BDD subgraphs, reducing the overall size of the BDD.

Notice that these two rules also accomplish the quad-tree partitioning shown in Figures 8 and 9, leading to improved compression. Section 6 shows that ParaMeter’s variable ordering yields good compression, indistinguishable from that of Price and Vachharajani’s original work [14].

5.3 ParaMeter DINxRDY Visualization

While BDDs are a natural fit for many analyses, especially data-dependence analyses and set manipulation operations, generating a DINxRDY plot is non-trivial due to the costs associated with standard BDD operations (e.g., and, or, not, etc.). To remedy this, ParaMeter directly traverses the BDD (treating it like a quad-tree) to quickly generate DINxRDY plots. This section describes the algorithm and the optimizations used to plot DINxRDY plots. The optimized algorithm runs in time that is proportional to the screen resolution, instead of the amount of data stored in the BDD (or trace). Section 6 quantifies the performance.

Figure 10 shows the pseudo-code for ParaMeter’s BDD-as-quad-tree DINxRDY plotting algorithm. The algorithm operates by mutual recursion, doing a depth-first traversal of the BDD, treating it as a partitioning tree that divides the region either vertically (by DIN) or horizontally (by RDY) at each recursive step. Every pair of calls to plotDINxRDY_din and plotDINxRDY_rdy corresponds to a traversal of one level in the quad tree. The isConstant() member function of BDD nodes returns true if the node is the constant node and false otherwise. The pixel member function plot() paints the pixel black. The region variable represents the screen region that corresponds to

the BDD sub-graph rooted at node. The region member functions divideByDIN() and divideByRDY() return a pair of new regions corresponding to the left (or bottom) and right (or top) halves of the region. The parity member function flip() returns a new parity object with the opposite parity.

There are three base cases.

1. Reached False

This case is illustrated in Figure 11(a). Here, we have reached the constant node in the BDD. If we reach the constant true node with odd parity, then we have reached false, which means that no tuples in the region region is in the DINxRDY set. Thus we should plot no points, and we simply return.

2. Reached True

This case is illustrated in Figure 11(b). If we reach the constant true node with even parity, it means that indicator function for the DINxRDY tuple-set indicates all tuples in the region region are in the set. Thus, we should paint the entire region.

3. Region Contained in Pixel

This case is illustrated in Figure 11(c). Here, the region corresponding to the BDD node node is entirely contained inside a pixel, and it contains useful data. If the region contained no data, node would be the constant node and we would have triggered the other base case.

Recall that large regions of most DINxRDY plots are empty. The performance of the traversal algorithm will depend on how early we can terminate the traversal of the BDD. For large empty regions, base case 1 is reached early and that part of the traversal ends quickly. At normal zoom levels, we quickly reach the region is entirely contained in a pixel (base case 3) and so these traversals also terminate quickly (in fact, the number of nodes traversed is on the order of black pixels to be painted, a number bound more by the number of pixels on the screen than DINxRDY data).

To handle high magnification levels, where traversal does not terminate quickly (and to improve performance in general), we extend the algorithm described in Figure 10 by terminating traversal of any BDD sub-graph that corresponds to a region outside the display area (commonly known as clipping in the graphics community). Traversing these regions is wasteful as they generate no visible output.

This optimization constrains the screen rendering performance to the screen size and the depth of the BDD (which is a constant 128 levels in ParaMeter). Thus, we can efficiently render the graph at high magnification levels, and our performance scales with screen resolution and size.

5.4 ParaMeter Slice Visualization

In addition to generating the DINxRDY visualization ParaMeter
The rectangular region can be described by two inequalities, one for the range on the DIN axis (call it the \( d \)-axis) and another for the range on the ready-time axis (call it the \( r \)-axis). If the selected DINxRDY rectangle’s lower-left corner is \((d_{\text{min}}, r_{\text{min}})\) and the upper-right corner is \((d_{\text{max}}, r_{\text{max}})\), the two inequalities are:

\[
\begin{align*}
    d_{\text{min}} & \leq d \leq d_{\text{max}} & (1) \\
    r_{\text{min}} & \leq r \leq r_{\text{max}} & (2)
\end{align*}
\]

Because the selected region may contain millions of entries, we need to extract a BDD for the selected DINs directly, without iterating over each DIN selected. This BDD will act as input to the slice computation algorithm in step 2.

Let \( L \) represent the vector of BDD variables for the \( r \)-axis and let \( D \) be the variables for the \( d \)-axis. Given the DINxRDY BDD (\( \text{BD}(\text{DINxRDY}) \)) and a BDD (\( I_A \)) corresponding to the user-selected region (\( A \)), we can extract the BDD for the selected DINs (\( I_S \)) via:

\[
I_S = \exists I_A : (I_A \land \text{BD}(\text{DINxRDY})) \tag{3}
\]

The \( \exists I_Q \) operation is called existential quantification, which can be performed rapidly on BDDs [4]. Existential quantification removes the \( y \) variables from a BDD in a manner such that if \( I_Q \) is an indicator for the set \( Q = \{ (x, y) \} \), \( \exists y I_Q \) results in the indicator function for \( \{ x : \text{there exists } y \text{ such that } (x, y) \in Q \} \).

To compute \( I_A \), we build two BDDs \( I_{A,d} \) and \( I_{A,r} \) such that \( I_A = I_{A,d} \land I_{A,r} \) where \( I_{A,d} \) is the BDD for the region of equation 1 and \( I_{A,r} \) is the BDD for the region of 2. Figure 12 shows pseudo-code for rapidly building these BDDs. The algorithm works by building two BDDs, \( \text{rl} \) for \( x \leq \text{upperB} \) and \( \text{ru} \) for \( x \geq \text{lowerB} \). It returns \( \text{rl} \land \text{ru} \).

To understand the inner loop that builds \( \text{rl} \) and \( \text{ru} \) consider only \( \text{ru} \). Here, we are building a BDD for the set \( \{ x : x \leq \text{upperB} \} \). We have the binary expansion:

\[
x = \sum_{i} x_i \cdot 2^i \leq \sum_{i} \text{upperB}_i \cdot 2^i = \text{upperB} \tag{4}
\]

The algorithm operates by induction on the \( i \)-th bit of \( x \) and upperB. When processing the \( i \)-th bit, the algorithm assumes that comparisons to all bits more significant than \( i \) were insufficient to decide if \( x \leq \text{upperB} \), so the \( i \)-th bit must be examined. From equation 4 we know that if all the bits more significant than \( x_i \) were insufficient to make a decision, then we have two cases.

\[
\begin{align*}
    x_i &= 1 - \text{In this case, if upperB}_i = 0, x \not\leq \text{upperB}. \text{ If upperB}_i = 1 \text{ we must look at the lower order bits.} \\
    x_i &= 0 - \text{In this case, if upperB}_i = 1, x \leq \text{upperB}. \text{ If upperB}_i = 0 \text{ we must look at the lower order bits.}
\end{align*}
\]

This case break-down is encoded in the \( \text{ru} \) BDD by

\[
\text{ru} = \text{ite}(x_i, \text{upperB}_i \land \text{ru}_\text{old} : \text{zero} : \text{ru}_\text{old})
\]

where \( \text{ru}_\text{old} \) means “look at the lower-order bits” because the loop iterates from LSb to MSb. \( \text{rl} \) is computed similarly.

Each BDD, \( I_{A,d} \) and \( I_{A,r} \), requires only 2n nodes where \( n \) is the number of bits in the representation for the DIN or ready-time. Since the BDD is built bottom up, from the lowest node in the order to the top node, only 2n node allocations are needed per BDD making construction fast.

We can optimize the computation in equation 3 by observing that (1) \( I_{A,d} \) does not involve ready times, (2) \( \exists I_1 \land I_2 = I_1 \land \exists x I_2 \) if \( I_2 \) does not contain the variables in \( x \), and (3) there is an optimized BDD algorithm to compute \( \exists x (I_1 \land I_2) \). Thus, the optimized version does

\[
I_S = I_{A,d} \land (\exists r (I_{A,r} \land I_{\text{DINxRDY}})) \tag{5}
\]

2. Computing Slices..

Given a set of DINs for which a dependence slice is to be computed, and given the data dependence graph as a BDD-compressed set of \( (\text{DIN}, \text{DIN}) \) tuples, the actual slice computation is straightforward; the computation corresponds directly to the standard image and preimage computations used in formal verifiers [3]. The basic algorithm is covered in the references [14]. Brayton et al. cover research on optimizing these computations [3].

3. Slice Visualization..

To visualize the results of slice analysis, we first extract the relevant sub-DINxRDY BDD by computing \( I_I = I_P \land I_{\text{DINxRDY}} \) where \( I_P \) is the result-set BDD computed by the slice algorithm and \( I_{\text{DINxRDY}} \) is the DINxRDY BDD. We plot \( I_I \) by running the ParaMeter DInxRDY plotter using a different symbol, color, or intensity. This leads to the plots in Figure 4.

BDDs can also be used for much more sophisticated analysis. For a discussion, readers are referred to the references [3, 20, 22]. Section 6.4 quantifies the performance of slice analysis and visualization.
5.5 Correlating to Source Code

To map a set of DINs back to the assembly code we compute

$$I_{PC} = \exists d. I_d \wedge I_{\langle DIN, PC\rangle}$$

where $I_d$ is a BDD for a set of DINs and $I_{\langle DIN, PC\rangle}$ is the BDD that maps DINs to PCs. With the PC set $I_{PC}$, standard techniques can map these PC locations back to the assembly code, and source code if debug information is available. Section 3 demonstrates the utility of this mapping.

6. QUANTITATIVE RESULTS

This section quantifies the performance of ParaMeter.

6.1 Evaluation Methodology

To evaluate the performance of ParaMeter, we collected data on its compression, slice analysis, and DINxRDY plotting performance. All comparisons are to Adamantium (Iyer et al.’s trace analysis tool [8]). All data was collected on a 2.8 GHz Pentium D workstation with 4 GB of RAM running Linux. The traces used for comparison are from gcc-compiled SPEC 2000 integer benchmarks run with the training input set. Results for ParaMeter use the CUDD [17] BDD package. We chose Adamantium as the baseline because it represents the current state of the art for DINxRDY plot visualization. Adamantium creates the DINxRDY plot by streaming through Burtscher-bzip2-compressed traces (see Section 4). ParaMeter uses the Price et al.’s [14] BDD representation and with the presented novel BDD variable ordering, and the presented novel algorithms.

6.2 BDD Space Efficiency

Figure 13 evaluates ParaMeter’s variable ordering by measuring compression (Figure 13). It shows the number of (DIN,DIN) data-dependence graph tuples that fit into a fixed amount of memory (512 MB, 1 GB, and 1.5 GB). Note that 254.gap is missing a value for 1.5 GB due to an infrastructure issue and that 256.gzip is not present due to excessively long trace collection times. Price and Vachharajani reported BDD encoded traces sizes of 12x to 60x less than a naïve representation [14]. The Figure 13 shows the new variable ordering used in this paper matches or exceeds the results presented in [14], demonstrating that ParaMeter’s variable ordering delivers sufficient compression. Even 181.mcf, which shows only linear growth, achieves over 50x compression at all sizes shown. Note further that with 1 GB of RAM we can store 500 million to 75 thousand startup trace data. The data shows that ParaMeter is 12 thousand to 75 thousand times faster than Adamantium. Results are similar across all window sizes that fit on modern monitors. The 254.gap (500) row in the table shows results for a 500 million instruction trace of 254.gap, which demonstrates the scalability of ParaMeter. Note that ParaMeter’s millisecond DINxRDY plot times are more than sufficient for panning, zooming, and other activities during interactive visualization.

6.4 Slice Analysis Performance

Figure 14 shows the performance of the novel aspects of the slice analysis routine: identification of DINs for a given user-selected region. The vertical axis measures the total time, in seconds, to create $I_A$ for a fixed region of the DINxRDY plot (recall that $I_A$ is the BDD for the user-selected set of DINs). Clearly, this creation time is well within bounds for interactive exploration. Price et al. report that Adamantium takes 1.8s just to find the 2 millionth instruction of the 175.vpr benchmark in minutes, without prior knowledge of the source code. The paper then describes how ParaMeter achieves

7. CONCLUSION

Though previous studies [2,8,10,13,19] show plentiful parallelism in sequential codes, finding exploitable thread-level parallelism (TLP) remains a daunting task. This paper shows how interactive visualization and analysis can help turn this potential parallelism into real TLP. The presented case study uses our new ParaMeter visualization tool to find exploitable data- and pipeline-parallel regions of the 175.vpr benchmark in minutes, without prior knowledge of the source code. The paper then describes how ParaMeter achieves
interactive performance for visualizations of large traces. The paper presents a new BDD variable ordering for BDD-compressed traces, novel algorithms for rapid plotting of DINxRDY plots, and algorithms for interactive slice-analysis and visualization. ParaMETER’s algorithms are sufficient for interactive use; full-screen visualizations render in under .02s, up to 75,000× faster than prior algorithms.

Acknowledgments

The authors thank the reviewers for their insightful comments, Fabio Somensi for his advice, Intel for support of this work, and Chinmay Ashok, Matthew Iyer, Josh Stone, and Neil Vachharajani for the Adamsantium framework. Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #NSF-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research (NCAR), and grant from the IBM SUR program, and the DRACO computer cluster. The ideas herein are not necessarily those of the above organizations.

8. REFERENCES


