Analyses for Extensive Parallelization of Video Applications in C

Shane Ryoo and Wen-mei W. Hwu

{sryoo,hwu}@crhc.uiuc.edu

IMPACT Technical Report

IMPACT-08-02

Department of Electrical and Computer Engineering and

Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

Initial public release: June 22nd, 2008

1 Introduction

With the introduction of multi-core microprocessors into mainstream computer systems, there is increasing pressure for performance-driven consumer applications to use this additional computational capacity. Unfortunately, few user applications effectively use these additional computing resources. Our previous work [1] has shown that it is possible for a compiler to identify a significant amount of coarse-grained (minimum of thousands of dynamic instructions) parallelism within a video encoder application, written in C. This is done using a combination of sophisticated, eager, whole-program analyses.

However, significant parallelism is still concealed from compilers. This is due to the lack of particular dataflow information that can only be acquired by analyses that must operate interprocedurally, through memory objects, and are believed to be inherently expensive. Although algorithms for obtaining some of this information have been previously published, they are generally not scalable to large programs or are restricted to programming languages less flexible than C. This is the primary obstacle for automated parallelization frameworks and tools for detecting parallelism today.

This technical report discusses some of the issues and obstacles we have observed in our efforts to automatically detect coarse-grained parallelism in video encoder applications written in C. We first describe the fundamental issues of obtaining dataflow information in a scalable manner. We then examine the additional analyses that allow the compiler to detect parallelism beyond that accomplished by [1]. We propose approaches for finding some of this information in a scalable manner. Finally, we discuss some of the obstacles impeding further development and use of these analyses.

2 Motivation

Compiler analyses have matured to the point where a compiler can begin to identify a significant degree of coarse-grained parallelism in programs with very regular behavior, such as video encoders. Such analyses must exceed a critical mass to be effective across more than a handful of small loops. This is because loop iterations and code regions must not have any dependences be-

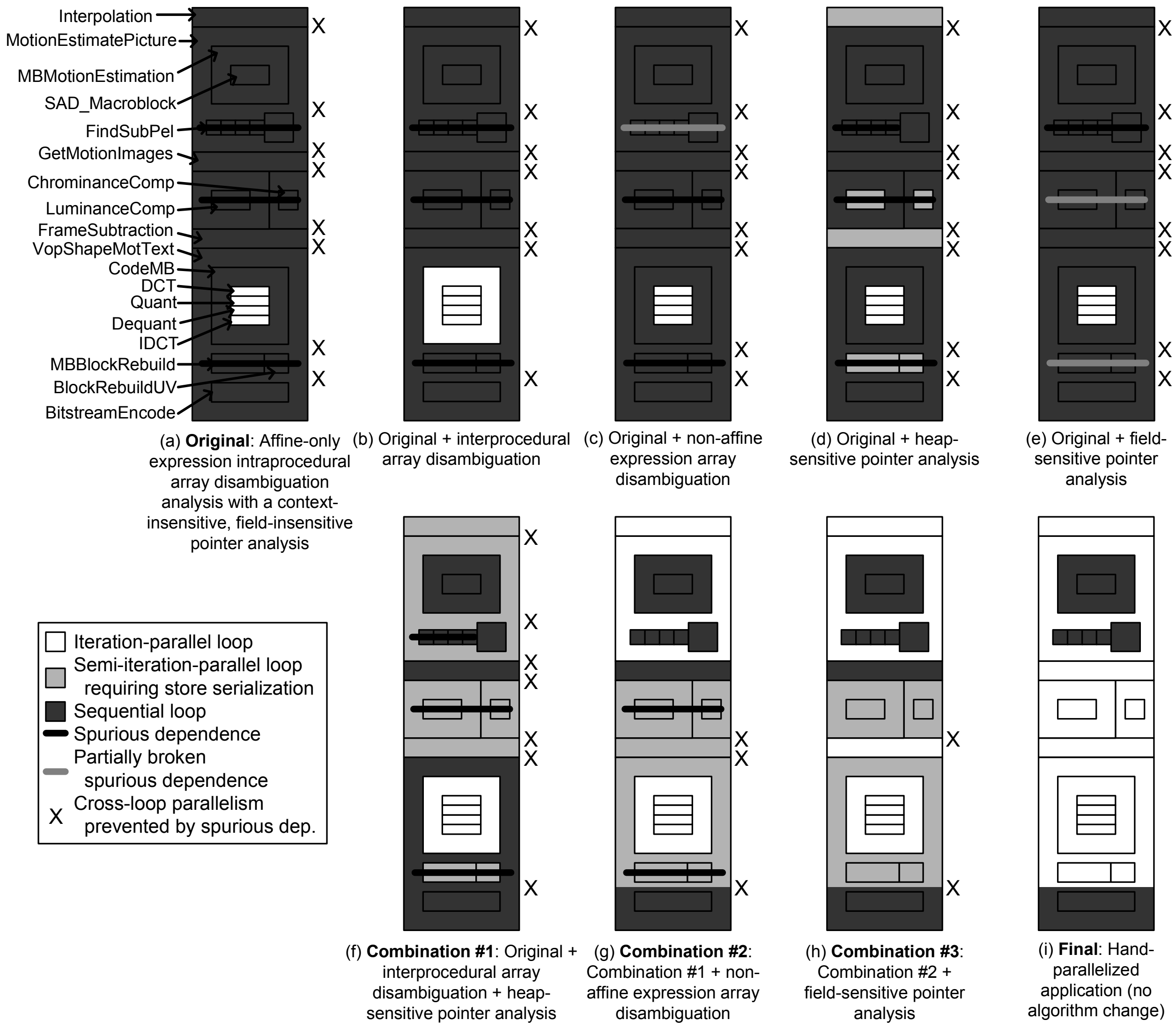

Figure 1: Effects of Various Analyses on the Compiler-Visible Parallelism in MPEG-4 P-frame Encoding

tween them in order to be completely parallel; if a single dependence appears to exist, some form of synchronization operation, speculation check, or code versioning is required to ensure program correctness. Also, various program constructs and coding styles necessitate particular analyses: if the analysis or analysis feature is not used in the presence of these code structures or programming patterns, then a parallelism detection opportunity is lost, which can have an extremely adverse effect on final performance. Although it is conceivable that a program developer could rewrite code to make it more analyzable, this requires understanding, either by the developer or by tools, of what coding patterns permit detection of parallelism.

We use a *loopblock* diagram, as shown in Figure 1, to visualize the parallelism within a portion of an MPEG-4 encoder application. The degree of parallelism observed depends on the sophistication of compiler analysis. Parts (a)-(h) show the parallelism revealed by different analyses or combinations of analyses, and (h) shows the parallelism that can be extracted by manually rewriting the application. Each block in the diagram represents a loop or nested loops that iterate over data structures. Completely contained blocks indicate loops nested within the outer blocks/loops. We observe three major forms of coarse-grained parallelism within this and similar media applications:

1. **Iteration parallelism**: An iteration of a loop can be executed independent of all other iterations. This is represented by the shading of the block: black shading represents a loop in which iterations must be run sequentially, while white indicates that each iteration of the loop is completely independent. There are situations where loop iterations can execute independently but stored values must be committed to global state in order because analyses cannot ensure that stores do not conflict with another; such parallelism is represented by gray shading in the figure.

   Dependences may be limited to a particular scope: although a loop's iterations may not be parallel, different instances of the loop can be independent. One example of this is instances of the `MBMotionEstimation` loop within iterations of the `MotionEstimatePicture` loop in Figure 1, parts (g), (h), and (i). In the `MBMotionEstimation` loop, iterations conditionally update a local variable based on the results of previous loop iterations. The loop interchange optimization [2] can create the same situation in other applications to increase the granularity of parallelism.

2. **Region parallelism**: Two or more static code regions can be executed in parallel. This is common when different static function calls (either to the same function or different functions) operate on different data or different sections of the same data object. This parallelism is represented by horizontally-adjacent blocks at the same scope/nesting level in the figure. Black horizontal lines through blocks represent obscured region parallelism. Gray lines represent parallelism that is blocked by fewer false dependences. Unlike iteration parallelism, the gray color here does not represent a partially exploitable form of parallelism. In Figure 1(c), the gray line through the instances of `FindSubPel` represents the fact that the

compiler has removed one obstacle to parallelism, which is determining that writes within the loops do not conflict with each other. In this case, the analyses cannot determine the independence of the loops' input and output arrays due to lack of heap-sensitivity.

3. **Cross-loop parallelism**: An iteration of a loop is dependent on a subset (often just one) of the iterations of the loop prior to it in program execution, and can begin execution when that subset of the previous loop's iterations have completed. Cross-loop parallelism is common in sequences of loops that operate on sections of an array. Some transformations that capitalize on this form of parallelism are software pipelining and loop fusion. In Figure 1 every loop interface at the same scope (having a horizontal interface) has potential cross-loop parallelism. Obscured parallelism is represented in the figure by an `X` on the right side of the shared boundary between loops.

Figure 1 shows the effects of program analyses, individually and in combination, on the compiler-visible parallelism in the P-frame encoding section of an MPEG-4 encoder application. Figures 1(a) through (e) show the minimal effects of a single analysis on visible parallelism. Figures 1(f) through (h) display how a combination of analyses expose a significant degree of parallelism. Figure 1(i) is a hand-parallelized version of the application without algorithm changes. With the exception of interprocedural array analysis, all of the analyses mentioned in the figure have been implemented in the OpenIMPACT compiler framework and run across a wide variety of benchmark applications. An explanation of the abilities and contributions of each analysis and analysis feature is detailed in [1].

In general, coarse-grained parallelism in media applications is detected by disambiguating memory objects and accesses to different locations within those objects. The first is usually accomplished by pointer analysis, while the second is accomplished by array analysis. It should be noted that different analyses usually are not biased towards exposing particular forms of parallelism. For example, heap objects can be used in any parallel situation, and a heap-sensitive analysis is probably necessary to identify any of those opportunities.

In comparing different parts of Figure 1, the reader may ask why the compiler or a developer should bother to try to achieve the level of parallelism shown in Figure 1(i) when (g) or (h) are already far better than (a). This question is justified because the required analyses are likely to be expensive to implement, integrate into a framework, and execute. The reason that exposure of this parallelism is desirable is that it is often when the last few restrictions are removed or optimizations performed that significant performance gains are achieved. For the application studied in Figure 1(i), the exposed parallelism allows for a loop fusion transformation on a grand scale. This would likely be an effective model of computation for a contemporary multicore processor or multiprocessor system, where inter-core/processor communication is expensive and local storage for instructions and data is plentiful. Figure 1(h) does not allow the same degree of freedom: execution must be divided into at least two loops, and all results from the first loop must be committed to global program state before the second loop can begin. This bottleneck in computation and communication may result in a significant performance detriment relative to what can be extracted. Thus, it can be desirable to use inherently expensive analyses, as long as they are targeted and controlled, because the gains can be substantial. In the next section we will describe some issues in constructing a scalable yet very precise analysis.

3 Analysis Issues

The total size of programs has steadily increased over the past decades. Because of the desire for maintainability, readability, and code modularity, this generally manifests as an increase in the number of procedures and how often they are called, rather than the size of procedures. Many of the practical analyses and optimizations used in production compilers are intraprocedural and thus have not been affected by the increased number of procedures. However, interprocedural analyses are necessary to find coarse-grained parallelism and must be carefully designed to function efficiently across programs of significant size. In this section we discuss the concerns of scalable interprocedural analysis designers and our approach to addressing them.

One of the most similar previous works on scalable analyses was performed at Microsoft Research for the purpose of program verification [3]. They describe many of the concepts discussed here in the context of using pointer analysis to reduce the size of dataflow problems. Unlike that work, we examine the combination of scalable analyses for obtaining information that would not be obtainable otherwise.

3.1 Scalability

As programs become larger, it is important to maintain the scalability of interprocedural analyses to ensure that they run in a reasonable amount of time and use a practical amount of space. The term "scalable" in this context means that the algorithm's runtime and space requirements in practice do not grow significantly faster than (i.e., roughly linear with) the written size of the program. Algorithms that are inherently and unavoidably exponential or high-degree polynomial in either time or space are generally not scalable, since they grow much faster than the size of the program and are generally unusable for large programs. We will discuss the impact that various program analysis features have on scalability when using this definition.

An example of scalable analysis is Erik Nystrom's work on the Fulcra pointer analysis framework [4]. The common misconception of inclusion-based, context-sensitive pointer analyses is that they are too expensive to run even on moderately-sized programs. For example, the inclusion-based pointer analysis previously used in the IMPACT compiler took several hours to complete when analyzing the 176.gcc benchmark from the SPEC CPU2000 suite. It was never run to completion for the 253.perlbmk benchmark due to its enormous space requirements for that benchmark. However, by pruning the context-sensitive data being copied into function summaries and performing other enhancements, Nystrom reduced the analysis time to under three minutes for both of these benchmarks without any loss in analysis precision. This approach to pointer analysis makes inclusion-based, context-sensitive pointer analysis usable for a larger set of programs than was previously thought feasible by the compiler community.

Many practical program analyses are polynomial approximation algorithms that make tradeoffs in order to obtain information in a fast and space-efficient manner. Algorithms that achieve more precision often take more time and space, and must be carefully constructed to enable scaling to larger programs. This can be done by taking advantage of traversal sparseness, data sparseness, and controllability.

1. **Traversal sparseness** consists of two items. The first is to utilize summaries of analysis structures to speed analysis. One common example is the use of transfer functions to represent the behavior of subroutine calls in a program. The second is to visit only analysis data structures that need updating and propagation, and not be required to check large code re-

gions/structures to see if an update is required. A *traversal-scalable* analysis takes advantage of traversal sparseness.

2. **Data sparseness** also consists of two items that have synergistic effects. The first is to use a compact representation that efficiently summarizes the analysis information at a program point. The second is to attach information only where it is relevant and necessary (i.e., when the previous information is no longer valid or is overly pessimistic). The classic example of these having a synergistic effect is reaching definition analysis using Single Static Assignment (SSA) Form [5]. In this form, the definitions of operands are explicitly encoded in versions or subscripts on the operands, and no information about irrelevant definitions is retained at an operation. A more dense implementation might tracking valid definitions of every variable at every program point. This scheme would store information about defintions at locations where they are irrelevant, and also retain storage space for definitions even at points where they would not be used. Effective use of data sparseness is the hallmark of a *data-scalable* analysis.

3. **Controllability** is necessary when certain code constructs or patterns inherently do not permit both precise and scalable analysis. In these cases, a choice must be made to give up some degree of one in favor of the other. Because the program code being analyzed may have global or near-global effects, decisions to control the analysis can have a significant impact on its precision. These effects may not be immediately obvious to the analysis designer or to the consumer of the analyses, such as a compiler.

   One common example of this is the decision to start the analysis from an optimistic state or a pessimistic state: optimistic methods must run to completion before having a correctly conservative state, while pessimism assumes a worst-case situation and attempts to refine it. Although pessimism provides high controllability by allowing the analysis to terminate at any time and still provide correct, conservative results, in some cases the results cannot achieve the same precision as the results possible with an optimistic starting state.

It is important to note that taking advantage of these aspects of analysis often does not improve the theoretical, worst-case time or space complexity of a particular analysis. However, in terms of real time and space requirements on real applications, the difference can be extremely signficant.

In practice, a theoretically faster algorithm may not complete execution on contemporary systems while scalable analyses complete in minutes.

In the previously cited example, one of the major scalability issues in Nystrom's analysis was the context-sensitivity feature: this replicates context information for each callsite, increasing the precision of the analysis at the cost of increased data size of the analysis. With large programs, this size can potentially increase exponentially. However, in practical programs, the analysis can prune information from upwardly propagated summaries when they have no effects at higher callgraph levels, constraining the analysis data size. Although pruning does not affect theoretical worst-case space requirements (it is theoretically possible to construct code that cannot be pruned) and can increase the amount of processing required by the analysis, it renders the analysis time and space for large benchmark applications scalable in practice.

3.2 Sensitivities of dataflow analysis

A dataflow analysis tracks the flow of data in and out of program objects/variables. Many optimizing compiler analyses are dataflow analyses; for example, live variable, reaching definition, available expression, and pointer analyses are all common dataflow analyses used in contemporary optimizing compilers. The flow of data is generally tracked via the names of variables or objects that are being read and written. For the purpose of detecting parallelism, the varying levels and aspects of precision, or sensitivities, can greatly affect the outcome of the final result. We discuss some of the sensitivities useful in discovering parallelism here.

A *flow-sensitive* analysis is distinguished from a flow-insensitive analysis in that it takes into account the order of operations in the program [6]. Flow-sensitivity not only eliminates situations that cannot occur due to the order of operations, but also allows the discovery of analysis information specific to particular program points. It also represents a potential trap for scalability, since a large amount of data may be stored at each program location. If the size of this data is not tightly constrained, the analysis data size may grow much faster than the size of the program. For this reason, it is common to make tradeoffs between precision and the speed or size of the analysis when performing a flow-sensitive analysis.

An extension of flow-sensitivity is *path-sensitivity*, where information for separate paths through the program are maintained. In more traditional dataflow analyses, path-specific information is often discarded. Because the total number of possible paths in a non-trivial program is effectively infinite, path-sensitivity is generally at odds with the desire for scalability. We do not discuss path-sensitivity in depth because previous experience has shown that the limitations required to make general, eager path-sensitive algorithms scalable often causes the results to be too conservative to provide significant benefit over an analysis that is not path-sensitive. Targeted analyses may be possible, and some of the analyses we discuss contain elements of path-sensitivity.

Context-sensitivity, in which the calling contexts/history of functions are distinguished, is an orthogonal issue to flow-sensitivity [7]. It provides increased precision by separately maintaining information of different function calling contexts via some degree of replication. Context-sensitivity also contributes to the scalability issue, primarily because the increased modularity and number of functions used in contemporary applications can grow the number of contexts at an exponential rate. The combination of flow- and context-sensitivity has a compounding effect that can create more scalability issues for the analysis designer.

4 Alternatives to Analysis

Given the number of dataflow analyses involved and potential scalability issues when run with high levels of sensitivity, it may be easier to obtain the information from the human developer rather than extract it from the code. This section discusses some of the alternatives to automatic analysis and why they may or may not be suitable for the purpose of detecting parallelism in video encoders.

4.1 Directives and Pragmas

Directives (Fortran) and pragmas (C) have often been used by developers to provide compilers information about the program that assists in optimization or transformation. Possibly the most commonly-known parallelism directive is the *doall* loop annotation in Fortran, which indicates that each iteration of a loop can be executed independently from every other iteration of that loop. More recently, the OpenMP API [8] provides developers with the ability to mark iteration-parallel loops and parallel code regions using pragmas to indicate their properties.

Pragmas are useful and currently necessary to extract parallelism from programs written with sequential semantics. However, they have the inherent limitations of being dependent on known parallelism and requiring the separation of serial and parallel code regions. The first is problematic because applications are often very large and few, if any, of the application's developers have correct or complete knowledge about all opportunities for parallel execution possible in the system. The second is an issue when an algorithm is implemented in a way that is inherently parallel but has a deeply buried serial section. Code is generally written to be readable and maintainable, not to distinguish serial and parallel sections. Separating the two may be difficult to do manually, even when assisted by tools. A case encountered in an MPEG-4 encoder application required over a week of effort to successfully extract the serial AC/DC prediction code, represented by the lowest black block in Figure 1, from the rest of the variable-length encoding section of the application. Without this transformation, the entire `CodeMB` block would be black, representing a significant decrease in the parallelism of the application. The final issue with pragmas is that code which is correct when executed serially may not be correct with the pragmas, and few mechanisms exist that can check whether a pragma maintains correct program execution.

4.2 Domain-Specific Languages and Extensions

Domain-specific languages such as [9, 10] have been proposed by multiple researchers as a means of expressing applications in a way that is more intuitive than C for the domains they target. They often restrict the use of pointers, require well-defined array boundaries, and otherwise prevent the use of code that compilers have traditionally found difficult to analyze. In doing so they provide a much easier context for the compiler to do its work, due to the assumptions inherent to the language or domain. Another benefit of domain-specific languages is that they can also increase developer productivity, since commonly used features are condensed into program statements with well-known behavior. Possibly the most common form of this paradigm is the use of Matlab by many scientists, where it is generally simple to express matrix operations and transforms.

The first issue with domain-specific languages or extensions is that developer productivity is usually the primary concern, with analysis being secondary. Productivity usually involves high modularity, ease of use, and access for debugging. These concerns may be at odds with analyzability or high performance of the applications. Previous work in telescoping languages has observed this effect [11].

Second, a domain-specific language or extension is useful as long as it is used in the well-defined domain for which it is intended. A developer may be operating in a new domain or experimenting with new algorithms, which violates this assumption. For example, during recent collaboration with MRI reseachers, we have encountered cases where the ubiquitous discrete cosine transform (DCT) was not applicable and transformations had to be expressed as more basic operations. In these cases, existing domain-specific languages may be unable to provide the proper support to developers and compilers must make worst-case assumptions about dependence information. Developing and supporting the specifications for a new language plus the required compiler support is generally an expensive and time-consuming proposition. Thus, domain-specific languages are not a complete solution to the parallelism detection problem.

4.3 Abstraction of Parallelism

In an approach related to the previous section, parallel code is provided to developers in the form of libraries. Libraries implement common functionalities, such as sorting, and hide the mechanics of managing parallelism, such as threads and locks. Some libraries, such as ScaLAPACK [12], provide multiple parallel algorithms. These packages usually allow the developer to choose the appropriate implementation for their application. Similarly, work on telescoping langauges [11] attempts to generate multiple versions of routines for high-performance libraries for domain-specific languages. This approach is now seeing more widespread usage and is gaining momentum.

Another form of parallelism abstraction is parallelism skeletons combined with thread-safe objects, such as Intel's Threading Building Blocks (TBB) [13]. Unlike a library, the developer must be aware of parallelism, but the burden of handling parallelism mechanisms is displaced to TBB; the developer only needs to write the code that operatess on each data object. NVIDIA's Compute Unified Device Architecture (CUDA) [14] has a similar structure, where the developer writes a single piece of code that is executed for all threads of a parallel kernel.

Ultimately, we envision library functionality being provided as a relatively atomic operation by the compiler to the developer. For example, a sorting routine would be provided to the developer as a library call. The compiler is provided dependence information and multiple versions of code with the library; its responsibility is to choose the version that fits well with the parallelism in the rest of the program and manage it accordingly. We have seen during previous work that poor matching of algorithms can have major effects on the overall parallelism of the application; providing multiple algorithms is one method to avoid such mismatches. In addition, encapsulation of parallelism in a library or parallelism skeleton removes the need for the developer to handle the mechanics of parallelism, such as threading, locking, and message passing. These have generally been error-prone areas of parallel program development. An extension of this idea is for the developer to provide multiple versions of certain code that the compiler can select from when compiling for a target platform. These types of approaches will be necessary for widespread development of parallel applications and are complementary to the analyses discussed here.

4.4 Interactive Tools

We draw a distinction between pragmas, which are generally placed within code prior to compilation, and developer-interactive tools, which we see as a future integral part of the compilation process. In the past, compilers were often treated as "black boxes" with a few external control knobs. However, the compiler's fundamental job is to make the software developer's work easier by managing the complexity of optimizing and translating a functional description of an application to a format that the target platform can execute. Having tools that allow the developer to use his knowledge and experience to control the compilation/synthesis process will make design cycles faster and less onerous to the developer. Analyses can be used as tools themselves, or can link into tools to inform the developer why code regions do not appear to be parallel to the compiler when the developer knows it is parallel. One example of this is work by Thies et al., which provides analyses and tools for exploiting pipeline-style, cross-loop parallelism [15].

5 Analyses

To expose the remaining parallelism shown in Figure 1(i), analyses other than ones discussed in [1] are required. In general, parallelizable work in video applications is performed on multidimension arrays. These arrays are stored in pointed-to data structures along with their sizes. The analyses discussed here address various aspects of these data structures. Here, we describe these analyses with the reasons they are needed and planned strategies for implementing them in a scalable manner. The analyses are:

- **Pointer/reference disambiguation**
  - The cycle sensitivity relationship, where data objects are passed among pointers every iteration of a program loop. The relationship between objects is obscured by the static code representation. We have developed a relatively scalable analysis that works in many cases [16].
  - Tree-relationships of linked structures, described in Section 5.2.
- **Array disambiguation**
  - Multidimension array disambiguation, as described in Section 5.3.
  - Memory dataflow analysis. This is necessary to privatize many variables, particularly globals used to pass data between functions.
- **Variable value analysis**
  - Interprocedural constant propagation, described in Section 5.5.
  - Variable value constraints relative to constants, described in Section 5.5.
  - Relationships between variables' values, described in Section 5.6.

5.1 Value flow through non-local memory objects

An issue that arises in whole-program analyses is the tracking of values and other information through non-local memory objects, such as dynamically-allocated objects and global objects. Attempting to incorporate them into a representation that specifies interprocedural dataflow is intractable [17] and most likely unscalable for programs with significant non-local memory. The majority of interesting values in media programs, such as array dimensions and loop bounds, are loaded from non-local memory because they are a part of structures that are dynamically allocated.

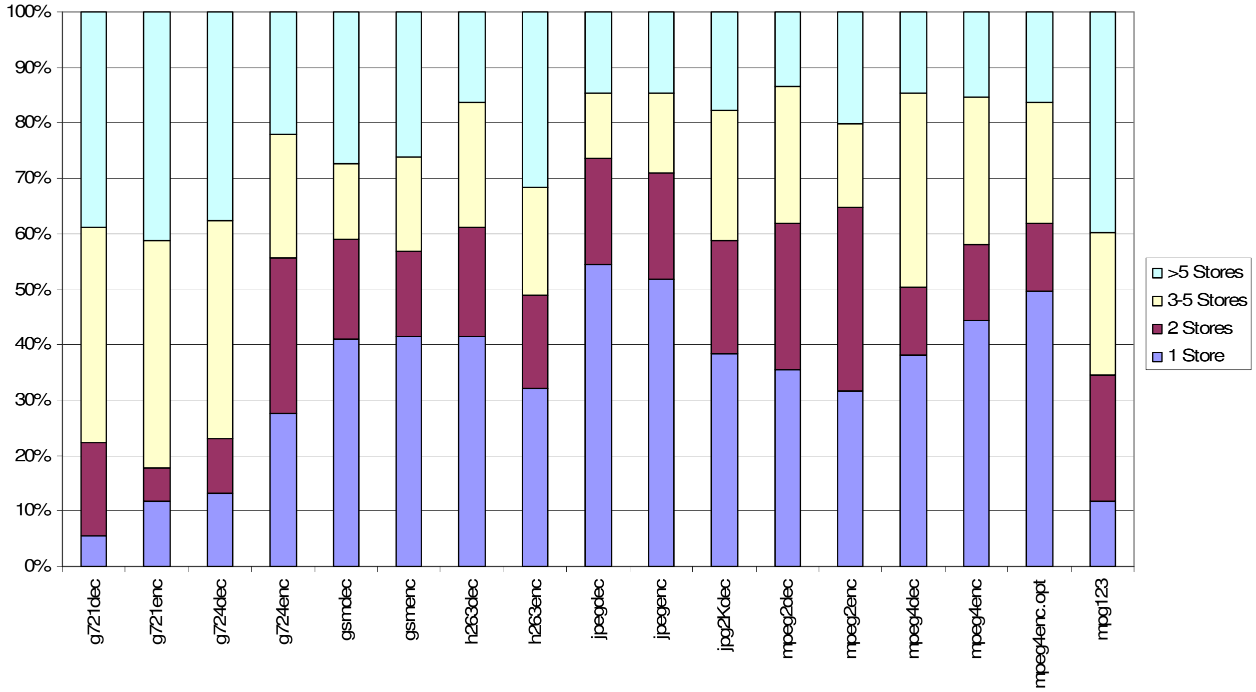

Figure 2: Static Store Operations per Object Field in Media Applications

To expose additional parallelism, it is necessary to track the flow of values through non-local memory objects, even though doing so in a precise, flow-sensitive manner is not generally feasible.

One approach for dealing with these values is to treat them flow-insensitively: all stores to one or more fields of an object are considered possible definitions of the memory location, and can reach any load of that field. This view is termed a program slice [18] of that memory load. This approach is practical because there are generally one or very few static store locations that can write many of those locations in highly data-parallel applications. Because of this, a flow-insensitive approach may be reasonably accurate. Previous work has used the same flow-insensitive strategy for dataflow problems pertaining to program verification [3].

Figure 2 shows, of the total distinguishable memory object fields in various media and telecom applications, the percentage of fields that have one or few store sites as distinguished by the Fulcra pointer analysis framework. Object fields with no store sites are dead memory objects and have been eliminated from the figure. This shows that treating memory objects flow-insensitively may allow reasonably precise dataflow results. One of the primary reasons this is possible is because the underlying pointer analysis is highly precise and can disambiguate a vast number of objects within the program. Attempting flow-insensitive memory tracking with less precision in field-sensitivity or heap-sensitivity may not allow useful dataflow analysis through nonlocal memory.

One potential issue is that some store sites are actually wrapper functions whose sole purpose is to write a field of an object. This may be a common occurrence for C++ applications when templates are utilized. The compiler would have to deal with numerous possible paths when it attempts to track values back through the various calling contexts. It may be possible to prune some unreachable contexts or inapplicable templates, but we have not established the methodology for this approach and it would likely depend on the analysis being run.

We anticipate that partial flow sensitivity using an interprocedural reachability analysis can aid in pruning store sites that do not reach a given load operation. For example, some applications clear fields of memory objects just prior to deallocation of the object for debugging purposes. If the compiler detects that the store cannot reach loads of the same memory location, then it can eliminate the store when performing dataflow analysis with the object.

5.2 Relational analysis

Although the iteration disambiguation analysis described in [16] is capable of distinguishing objects created in separate iterations of a loop, this is insufficient for parallelizing most media applications. These disambiguated objects are usually aggregate objects that contain pointers to arrays or other child objects that hold the actual media data. Iteration disambiguation [16] fails with these objects because they are created and linked to the aggregate object (escaping what can be tracked by the analysis) and loaded when used (as an ambiguous reference). In MPEG-4, the aggregate frame object, or `Vop`, has references to three `Image` objects, one for luminance and two for chrominance. Parallelization of the reference MPEG-4 encoder requires that the compiler know that each `Image` object belongs only to a single `Vop`, which both iteration disambiguation and traditional pointer analysis cannot provide.

To supplement iteration disambiguation, the compiler will need to understand that each aggregate object receives independent child objects when it is created. An analysis must also be able to determine that child objects are never shared or swapped elsewhere in the program. This is a case related to that of disambiguating multidimension arrays, as discussed in the next section. Cases we observed do not have children objects swapped or moved to other aggregate objects,

```
int ***a, i, j, k;
a = malloc (x * sizeof (int **));
for (i = 0; i < x; i++)
  {
    a[i] = malloc (y * sizeof (int *));
    for (j = 0; j < y; j++)
      a[i][j] = malloc (z * sizeof (int));
  }

for (i = 0; i < x; i++)
  for (j = 0; j < y; j++)
    for (k = 0; k < z; k++)
      a[i][j][k] = init_value;
```

Dependence?

Figure 3: A Heap-allocated Multidimension Array.

which means that an analysis does not have to retain location-specific result information for these children objects and only needs to determine that the object's reference is only written to a single parent object upon creation.

Connection analysis [19] can find some limited cases of this situation: if the aggregate objects are disambiguated by the analysis, the child objects can also be disambiguated. However, child objects belonging to the same aggregate object, such as the different subframes of an image, cannot be disambiguated because they are connected. Similar situations have been addressed in shape analyses, some of which can handle arrays of references [20], but shape analyses have generally not been highly scalable.

5.3 Multidimension arrays

Traditional pointer analysis has issues when attempting to distinguish dynamically-allocated objects that come from the same call site and calling context. One common occurrence of this phenomenon is multidimension arrays. In the C language, a multidimension array can be declared as an allocated tree of pointers to arrays, shown in Figure 3. This structure must be identified as a multidimension array and not a more arbitrary structure before array access disambiguation analyses can be applied to it: the C language does not prevent different indices (`i` and `j` in the figure) from pointing to the same object. In Figure 1(f), lack of knowledge about multidimension arrays is one of the obstacles to the parallelization of `VopShapeMotText` block.

Some multidimension arrays are created interprocedurally. For example, in the SPEC CPU2006 benchmark 464.h264ref, an allocation call for a three-dimension array consists of multiple calls to an allocation function for a two-dimension array. Wu et al. [21] have an analysis that is capable of handling intraprocedural cases and have proposed a method for analyzing interprocedural ones. Venet [22] has a technique that achieves a similar effect, based on iterative solution of semantic equations. We believe the method to address this problem efficiently in the interprocedural case is the same as the relational analysis of the previous section.

5.4 Interprocedural constant propagation

Constant propagation is a well-recognized optimization that has shown to be useful across a wide range of applications domains. It operates by determining if the value assigned to a variable is a constant, then replaces reached uses with the constant. In general, constant propagation algorithms are inexpensive and can be run eagerly.

Interprocedural constant propagation, especially with function cloning, has been found to be useful in extracting parallelism in some benchmarks [23]. We anticipate that it will be effective for removing unreachable program paths that otherwise break opportunities for parallel execution, particularly error-checking paths. For example, the `GetMotionImages` loop in the MPEG-4 encoder from Figure 1 contains a check in the source code that may exit the loop early. Parallelizing this code as it is written requires speculative execution of loop iterations. However, the constant values assigned to the checked locations in the `MotionEstimatePicture` loop will never trigger the check. By removing this check and the early exit, the `GetMotionImages` loop can be parallelized without speculation.

There are many algorithms and options published for interprocedural constant propagation, a few of which are described in [23–27]. However, constant propagation functionality can be subsumed by a broader analysis and optimization that generalizes the reaching, constrained value property. This is described in the next subsection.

5.5 Value constraint analysis

The flows of values in programs have long been utilized for effective optimization, with examples ranging from constant propagation to partial redundancy elimination (PRE) and partial dead code elimination (PDE) [28–32]. These techniques are based around explicit computation and assignment of variables, and are well accepted by the compiler community, even if they are not always used.

During previous work, we encountered a different class of value relationships and optimizations, where values of variables are not specifically assigned constants or expressions, but were *constrained* in their possible values by dominating branches that compared the variables to other variables or constants, or by multiple instances of constants assigned to a single variable that merge in control flow. An example in the 255.vortex benchmark from the SPEC CPU2000 suite contains numerous redundant checks of a variable's value equal to zero: after the first check, the value of the variable is constrained to zero and later instances of the variable can be changed to zero. This removes a significant number of unreachable paths and reduces the number of executed instructions, resulting in improved hyperblock formation and several percent performance increase.

The existence of these redundant checks is due to code modularity. The same variable is passed to several functions, each of which checks its arguments to make sure it is called with valid inputs. In the case encountered above, inlining of the functions makes the checks redundant along an intraprocedural path of the program. The redundant checks were then removed by inserting a virtual constant write and performing a constant propagation optimization. Although value constraints that enable this optimization are not widespread in general-purpose programs such as the SPEC CPU integer benchmarks, value constraints have a long history of use in scientific computing and often appear in media applications that operate on a fixed granular size.

Value constraints can be critical in disambiguating array accesses and exposing parallel loop iterations. An example of the value constraint problem from the MPEG-4 encoder is seen in `GetPred_Default` in Figure 4. LOOP2 increments `index2` by `width` each iteration. If the absolute value of `width` is less than 16, then an output dependence exists between iterations of LOOP2, preventing independent execution of iterations of LOOP2, LOOP3, or LOOP4. If the compiler can trace the value back through the heap structures, it can determine that error check and alignment code restricts the value of the image width to an integer multiple of 16. Figure 5(a) shows the access conflicts the compiler must assume are possible without value constraint information for the value of `width`, versus the actual application behavior in Figure 5(b).

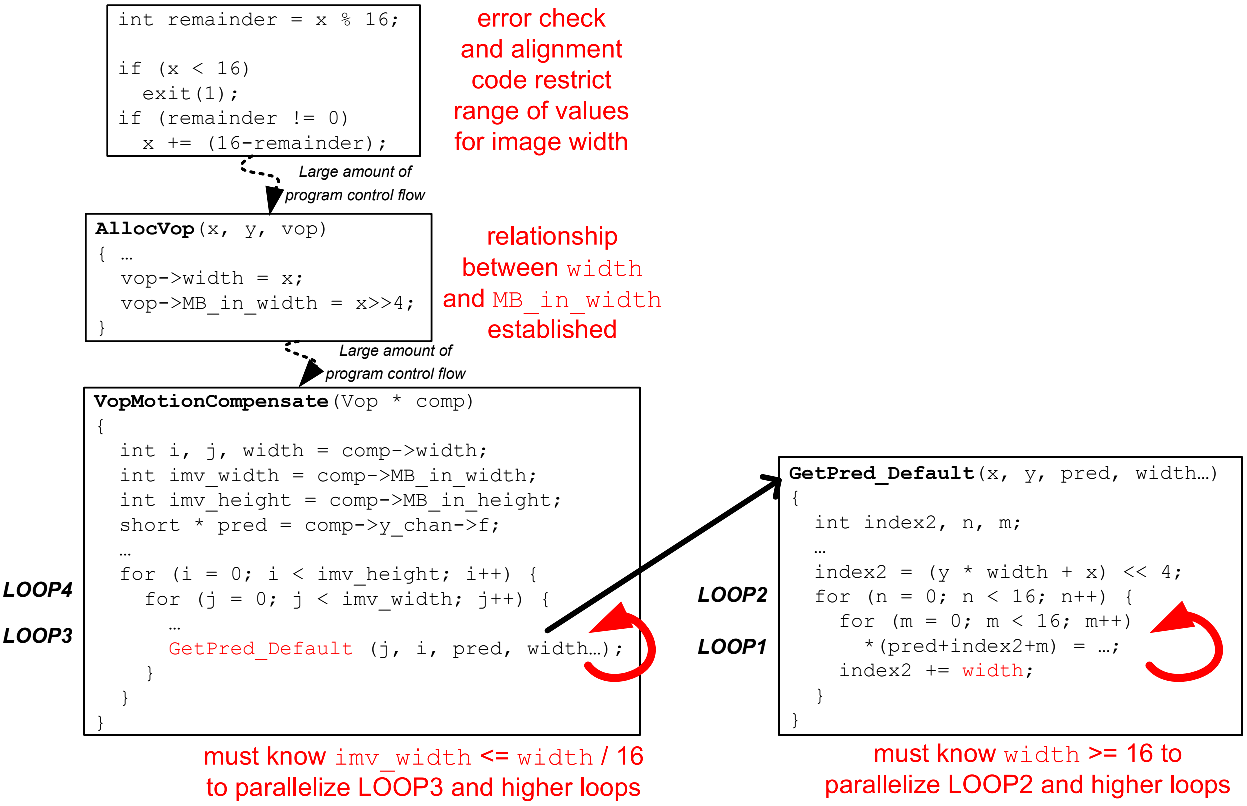

Figure 4: Example of the Value Constraint and Relationship Problems in MPEG-4 Motion Compensation

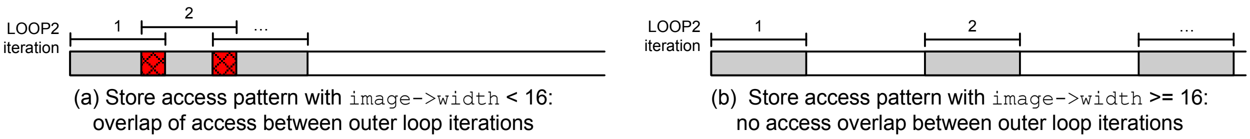

Figure 5: Access Patterns for the Value Constraint Problem in Figure 4

5.5.1 General applicability and previous work

Although exposing loop-iteration parallelism is possibly the most profitable use of value constraints, value constraints have more general usage as well. Possibly the most common use is removal of array bounds checks, which is targeted in the majority of previous work. Other examples include removing correlated branches and unreachable paths, which has been associated with PRE by Bodik et al. [33], and optimizations within large hyperblocks as a result of predicate analysis [34]. Restriction of values to a single constant allows constant propagation, as shown by the 255.vortex example, and has been implemented for intraprocedural cases in OpenIMPACT's classical optimization phase. It is also expected to be effective in specializing inlined calls for specific classes during OpenIMPACT's C++ to C compilation stage.

Some of the oldest previous work in this area has attempted to find value constraints in a dense dataflow fashion [35]. We anticipate that these will suffer from enormous growth and blocked opportunities at dynamically-allocated objects when dealing with today's applications written in C. Much previous work is based on formal methods, such as the initial work on abstract interpretation [36–38], and may not scale well to moderately sized programs. A demand-driven analysis by Blume and Eigenmann [39] is likely to be usable for large programs, and an interprocedural version was more recently published [40]. The former work separates control and data constraints, but there may be an advantage in handling both simultanously. The majority of the prior work is focused on Fortran programs, whereas we are interested in the predominant contemporary languages, such as the C language and its derivatives.

More recent related work includes methods for handling pointer arithmetic and conditionally updated induction variables [41], although they do not extend their algorithm to interprocedural cases. Patterson uses range analysis to perform branch prediction [42] and Verbrugge et al. [43] perform it interprocedurally for more general purposes, although both are limited to non-symbolic constraints. Other work has performed value constraint analysis for specific cases, such as eliminating virtual function calls in C++ [44].

5.5.2 Proposed algorithm

We believe a viable approach to intraprocedural value flow is a dataflow approach similar to that performed by Patterson [42]. First, critical edges as defined in [45] are split to ensure that every branch has a control-equivalent block. At each branch path the compiler inserts a new definition of the variables tested in the branch. This creates SSA-style def-use chains for every constraining operation in the program, whether it be via explicit assignment or constrainment via branches. Tracing explicit constrainment of variables is then a relatively simple process as long as the values flow through local variables, parameters, and function return values and there are not too many values of interest. An example is shown in Figure 6. OpenIMPACT currently performs this analysis for branches where variables are compared equal or not equal to a constant.

A critical issue with whole-program value-flow information is how to deal with values flowing through non-local memory objects, as discussed in Section 5.1. In an ideal dataflow environment,

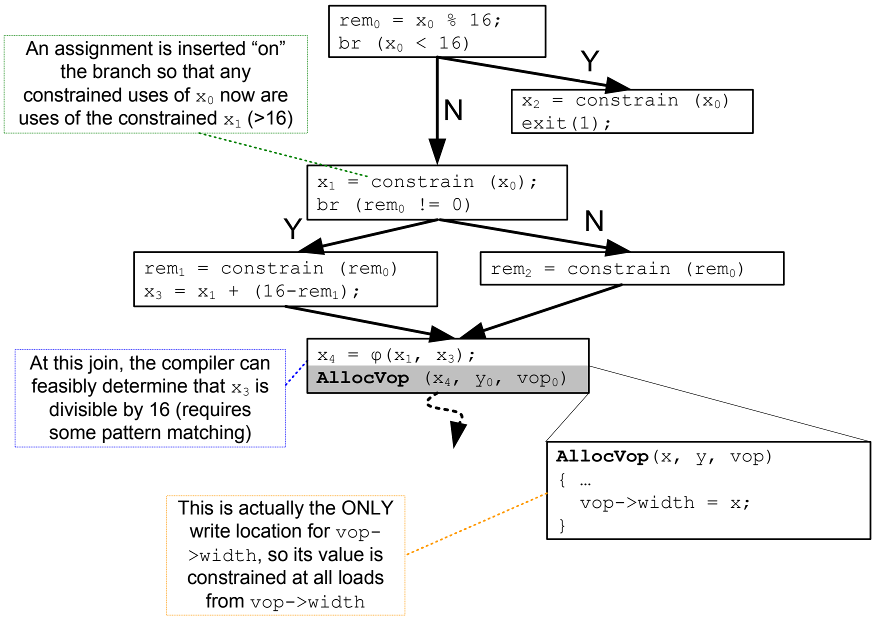

Figure 6: Example of Value Constraint in Action

the compiler must know which write instances or branch paths can reach a particular use. This is complicated by the presence of multiple procedure calls, which can force conservative behavior and include false reaching paths. A scalable approach is to rely on pointer analysis to provide precise information about the total number of static accesses that may write a given object or field of an object. The analysis can take a flow-insensitive approach and treat all of them as valid, as discussed in Section 5.1.

Another issue is values that are not explicitly constrained by a constant, but are related to a constrained variable at a particular program point and thus inherit constraints. For example, a code segment could assign the value of a variable `x` to a variable `y` and then check if the value of `x` is equal to zero. The compiler does not have knowledge of `x`'s value with the proposed algorithm. Attempting to analyze such information is an unscalable prospect: the potential number of relationships between variables has no reasonable limit for non-trivial programs. Knowledge of the relationships between variables can be valuable, though, and we discuss how to obtain this information in the next subsection.

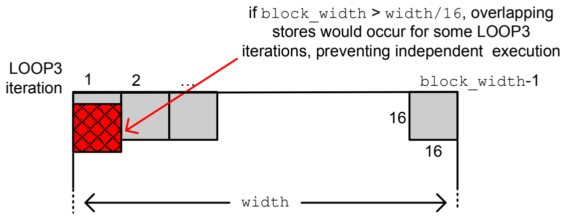

Figure 7: Store Access Pattern for LOOP3 in Figure 4

5.6 Value relationship inference

Array disambiguation analyses sometimes require knowledge of the relationships *between* variables that are used in loop bounds and array index expressions. The compiler can find these relationships by tracking values back through def-use relationships to find constants and common terms. Although OpenIMPACT currently performs this back-tracking intraprocedurally, it cannot address situations where the values are stored in non-local memory objects.

A common example of value relationships in the MPEG-4 encoder is the constant ratio between the width of an image in pels and (macro)blocks. When both of these appear in an array disambiguation test, the compiler must be able to find their relationship. This example is shown in Figure 4. The relationship between the variables is established in `AllocVop`. In `VopMotionCompensate`, LOOP3 has an upper limit of `imv_width-1`. The value of `j` is passed to `x` in `GetPred_Default`. If `j` is equal to or greater than `width/16`, then an output dependence occurs across calls to `GetPred_Default`, and iterations of LOOP3 and LOOP4 cannot be independently executed. However, since the maximum value of `j` is `(imv_width-1)`, or `(width/16-16)`, no output dependence exists across iterations of LOOP3 or LOOP4.

Figure 7 shows the effects of the lack of value relationship information on the target image. Each call to `GetPred_Default` fills in a 16x16 pel block in the image. Without value relationship information, the compiler must assume that the write of the hashed block is possible and must refrain from storing the results of loop iterations in parallel. Value relationship information tells the compiler that this access is not possible, so parallel store commits are legal.

Unlike the value constraint problem discussed in Section 5.5, this problem deals with the relationships of program variables with each other as well as numeric constants. Many of the previous works mentioned in that section attempt to solve this problem, but most of them have been based on eager analyses that grow exponentially with program size and are inherently unscalable. However, Tu and Padua have published a demand-driven algorithm using Gated Single Assignment form [46] and a related interprocedural method has been published by Bae and Eigenmann [40]. The latter analysis is controllable, which may be required for a practical value relationship analysis. We believe a practical implementation can be based on this method, extended to pass values through non-local memory in a flow-insensitive manner as discussed in Section 5.1.

6 Obstacles

We encountered two major obstacles to parallelism detection during our work on discovering and extracting parallelism. These issues must be addressed before more work in parallelism-detecting analysis can be effective. They are discussed here.

6.1 Multidimension array identification and optimization

In many media applications, multidimension arrays are often implemented as structures that are not trees of pointers. For example, in the MPEG-4 encoder used in [1] as well as the 464.h264ref benchmark from SPEC CPU2006, two-dimension arrays are flattened into single-dimension arrays and the burden of correct address calculation is placed on the developer. Both structures are implemented to capture two-dimension locality of the video images. The MPEG-4 code uses a single flattened array, while the 464.h264ref code implements an array of pointers that points to the beginning of rows within a separate flattened array.

Unfortunately, flattening a multidimension array negatively impacts the ability of a compiler to analyze the code and may also cloud a developer's view of the code. The situations found in Figures 5(a) and 7 can only happen with flattened multidimension arrays. Attempting to prove independent accesses with value constraint and value relationship analyses is generally difficult, assuming that all corner cases have been properly bounded. If a compiler understood that the structures being accessed were multidimension arrays and had knowledge about their dimensions (probably as an explicit part of the structure), the need for value constraint or value relationship information would be reduced or eliminated and make parallelism detection less complicated.

One method of addressing this situation is to provide compiler support for a a single expression for a multidimension array backed by the ability to choose one of a variety of implementations.

This choice can be made automatically by the compiler or annotated on the code by the developer. This format results in more portable and readable code and reduces the amount of analysis required to prove that loop iterations are parallel.

Related work that addresses a similar issue, that of array tiling, is Hierarchically Tiled Arrays (HTA) [47]. These are arrays that are subdivided into tiles, used in MatLab and C++, to express locality and parallelism in a cleaner manner. Follow-on work [48–50] has shown the usefulness and useability of these structures.

6.2 Code correctness

Possibly the largest obstacle to automatic parallelization of code is the correctness of applications. Specifically, many applications do not successfully restrict incoming data values to a legal subset before entering highly parallel code regions. This issue was present for several MPEG-4 encoder applications that we studied. In Figure 4, the checks shown in the uppermost blocks would be either absent or incomplete, allowing conditions such as image widths that are negative or not multiples of 16 pels. When invalid inputs are given to the program, these missing checks allow access patterns like those in Figures 5(a) and 7, preventing parallelism detection. Although a production application is likely to be more robust than the reference codes we studied, there are usually bugs in any significant application.

One possible solution to this problem is a language- or compiler-supported mechanism to annotate expected program properties onto codes. Such support is being developed for new programming languages or language extensions and advanced development environments, such as C Spec for Microsoft's C#. Even if these annotations were not enforced during runtime via assertions, at the very least they would provide hints to a compiler about the properties of the program and enable speculative parallelization without the need to profile the application. On the other hand, if the compiler can communicate why a region cannot be parallelized when the developer believes parallelization is possible, parallelism detection analyses can be a useful debugging tool.

7 Summary

This report has discussed some the necessary analyses required to detect nearly all of the coarse-grained parallelism present in many video encoder applications. We mentioned several scalability features of analyses as well as alternatives to analysis. We then describe the major analyses necessary for detecting the remaining parallelism in one video encoder application, along with examples of where they are used. Some of these analyses, particularly value constraints and relationships, are inherently expensive and must be tightly controlled to be practical.

Future work in parallelism detection is hindered by the fact that multidimension arrays can be expressed in several ways in C, some of which cause difficulties for parallelism detection. In addition, code may not sufficiently exclude invalid inputs, allowing situations where inherently parallel code has corner cases where serial execution is required to maintain program semantics. We believe that these problems can be mitigated with programming language extensions that improve developer productivity and the correctness of code.

Acknowledgments

We thank the other members of the IMPACT research group for their support. This work was supported by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W. W. Hwu, "Automatic discovery of coarse-grained parallelism in media applications," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 1, no. 1, pp. 194–213, 2007.

[2] J. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction*, June 1984, pp. 233–246.

[3] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer, "Speeding up dataflow analysis using flow-insensitive pointer analysis," in *Lecture Notes in Computer Science*. Springer-Verlag, 2002, vol. 2477, pp. 230–246, proceedings of the 9th International Symposium on Static Analysis.

[4] E. M. Nystrom, "FULCRA pointer analysis framework," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1998, pp. 1–11.

[6] J. D. Choi, M. G. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, January 1993, pp. 232–245.

[7] T. J. Marlowe, B. G. Ryder, and M. G. Burke, "Defining flow sensitivity in data flow problems," manuscript, dated July 17, 1995.

[8] OpenMP Architecture Review Board, "OpenMP application program interface," May 2005.

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 2002 International Conference on Compiler Construction*, 2002, pp. 179–196.

[10] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 42–54, September/October 2004.

[11] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellow-Crummey, "Telescoping languages: A system for automatic generation of domain languages," *Proceedings of the IEEE*, vol. 93, pp. 387–408, February 2005.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[13] Intel Technical Staff, April 2008. [Online]. Available: http://www.intel.com/software/products/tbb.

[14] NVIDIA Technical Staff, "NVIDIA CUDA," http://www.nvidia.com/cuda.

[15] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the 40th International Symposium on Microarchitecture*, December 2007, pp. 356–368.

[16] S. Ryoo, C. I. Rodrigues, and W. W. Hwu, "Iteration disambiguation for parallelism identification in time-sliced applications," in *the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[17] E. W. Myers, "A precise inter-procedural data flow algorithm," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, January 1981, pp. 219–230.

[18] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[19] R. Ghiya and L. J. Hendren, "Connection analysis: A practical interprocedural heap analysis for C," in *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, August 1995, pp. 515–533.

[20] F. Corbera, R. Asenjo, and E. L. Zapata, "Towards compiler optimization of codes based on arrays of pointers," in *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing*, 2002, pp. 142–156.

[21] P. Wu, P. Feautrier, D. Padua, and Z. Sura, "Instance-wise points-to analysis for loop-based dependence testing," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 262–273.

[22] A. Venet, "A scalable nonuniform pointer analysis for embedded programs," in *Proceedings of the 11th Static Analysis Symposium*, August 2004, pp. 149–164.

[23] R. Metzger and S. Stroud, "Interprocedural constant propagation: An empirical study," *ACM Letters on Programming Langauges and Systems*, vol. 2, no. 1-4, pp. 213–232, March-December 1993.

[24] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 1985, pp. 291–299.

[25] D. Callahan, K. Cooper, , K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, 1986.

[26] D. Grove and L. Torczon, "Interprocedural constant propagation: A study of jump function implementations," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993, pp. 90–99.

[27] P. R. Carini and M. Hind, "Flow-sensitive interprocedural constant propagation," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995, pp. 23–31.

[28] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, vol. 22, pp. 96–103, February 1979.

[29] R. Bodik, R. Gupta, and M. L. Soffa, "Complete removal of redundant computation," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998, pp. 1–14.

[30] R. Bodik and S. Anik, "Path-sensitive value-flow analysis," in *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, January 1998, pp. 237–251.

[31] R. Bodik, "Path-sensitive, value-flow optimizations of programs," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, PA, 1999.

[32] S. Ryoo, "Partial code elimination in the IMPACT compiler framework," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 2004.

[33] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, June 1997, pp. 146–158.

[34] J. W. Sias, "Condition awareness support for predicate analysis and optimization," Master's thesis, University of Illinois, Urbana, IL, 1999.

[35] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. 3, no. 3, pp. 243–250, May 1977.

[36] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*, April 1976, pp. 106–130.

[37] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[38] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, 1978, pp. 84–96.

[39] W. Blume and R. Eigenmann, "Demand-driven, symbolic range propagation," in *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, 1995, pp. 141–160.

[40] H. Bae and R. Eigenmann, "Interprocedural symbolic range propagation for optimizing compilers," in *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, October 2005, pp. 340–346.

[41] J. Birch, R. van Engelen, and K. Gallivan, “Value range analysis of conditionally updated variables and pointers,” in *Proceedings of Compilers for Parallel Computing*, 2004, pp. 265–276.

[42] J. R. C. Patterson, “Accurate static branch prediction by value range propagation,” in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995, pp. 67–78.

[43] C. Verbrugge, P. Co, and L. J. Hendren, “Generalized constant propagation: A study in C,” in *Proceedings of the 6th International Conference on Compiler Construction*, 1996, pp. 74—90.

[44] G. Aigner and U. Hölzle, “Eliminating virtual function calls in C++ programs,” in *Proceedings of the 10th European Conference on Object-Oriented Programming*, 1996, pp. 142–166.

[45] J. Knoop, O. Rüthing, and B. Steffen, “Lazy code motion,” in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementaton*, June 1992, pp. 224–234.

[46] P. Tu and D. Padua, “Gated SSA-based demand-driven symbolic analysis for parallelizing compilers,” in *Proceedings of the 9th Interational Conference on Supercomputing*, July 1995, pp. 414–423.

[47] G. Almási, L. De Rose, B. B. Fraguela, J. Moreira, and D. Padua, “Programming for locality and parallelism with hierarchically tiled arrays,” in *the 16th International Workshop on Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science (LNCS) 2958, October 2003, pp. 162–176.

[48] G. Bikshandi, B. B. Fraguela, J. Guo, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua, “Implementation of parallel numerical algorithms using hierarchically tiled arrays,” in *the 17th International Workshop on Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science (LNCS) 3602, September 2004, pp. 87–101.

[49] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, “Programming for parallelism and locality with hierarchically tiled arrays,” in *Proceedings of the ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, March 2006, pp. 48–57.

[50] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. B. Fraguela, M. J. Garzarán, D. Padua, and L. Rauchwerger, “Design and use of htalib – a library for hierarchically tiled arrays,” in *the 19th International Workshop on Languages and Compilers for Parallel Computing*, November 2006.