Automatic Discovery of Coarse-Grained Parallelism in Media Applications

Shane Ryoo, Sain-Zee Ueng, Christopher I. Rodrigues, Robert E. Kidd, Matthew I. Frank, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{sryoo, ueng, cirodrig, rkidd, mif, hwu}@crhc.uiuc.edu

Abstract. With the increasing use of multi-core microprocessors and hardware accelerators in embedded media processing systems, there is an increasing need to discover coarse-grained parallelism in media applications written in C and C++. Common versions of these codes use a pointer-heavy, sequential programming model to implement algorithms with high levels of inherent parallelism. The lack of automated tools capable of discovering this parallelism has hampered the productivity of parallel programmers and application-specific hardware designers, as well as inhibited the development of automatic parallelizing compilers. Automatic discovery is challenging due to shifts in the prevalent programming languages, scalability problems of analysis techniques, and the lack of experimental research in combining the numerous analyses necessary to achieve a clear view of the relations among memory accesses in complex programs. This paper is based on a coherent prototype system designed to automatically find multiple levels of coarse-grained parallelism. It visits several of the key analyses that are necessary to discover parallelism in contemporary media applications, distinguishing those that perform satisfactorily at this time from those that do not yet have practical, scalable solutions. We show that, contrary to common belief, a compiler with a strong, synergistic portfolio of modern analysis capabilities can automatically discover a very substantial amount of coarse-grained parallelism in complex media applications such as an MPEG-4 encoder. These results suggest that an automatic coarse-grained parallelism discovery tool can be built to greatly enhance the software and hardware development processes of future embedded media processing systems.

1 Introduction

In the past few years, several multicore microprocessors have been introduced for both general purpose and embedded systems computing [1,5,19,20,30]. Despite the parallelism many contemporary applications exhibit, compilers currently do not have the capability to automatically extract substantial coarse-grain, thread-level parallelism from them. Explicit parallel programming by human programmers also remains a major challenge. One of the primary reasons for this dilemma

P. Stenström (Ed.): Transactions on HiPEAC I, LNCS 4050, pp. 194–213, 2007.

is that multiple, sophisticated interprocedural analyses, performed by either a human or a compiler, must achieve a clear view of memory usage before safe and profitable transformations can be performed. There is also a lack of understanding in how one can build an analysis system to facilitate hand and automated parallelization of full-fledged, pointer-heavy modern applications. The goal of this work was to investigate the degree of coarse-grained parallelism that can be automatically discovered by an analysis system and determine the state of the analyses required to expose that parallelism in contemporary applications.

For this work we chose to focus on the media application domain, which has a high degree of inherent parallelism, a large user base desiring higher performance, and available reference codes from industry standards bodies. These applications are often implemented in C/C++, which have features that require sophisticated analysis to disambiguate memory accesses: pointers, indirect procedure calls, dynamically-allocated memory, and resizable data arrays. Some of the programs we have investigated are reference MPEG-4 encoders from standards bodies, jpegdec from MediaBench, and the LAME and mpg123 applications for MPEG Layer 3 audio encoding/decoding. These codes are often referenced when programmers write explicitly parallel versions or when hardware designers transfer algorithms into hardware description languages; thus, parallelism discovery is useful as a tool for these designers. It is worth noting that the target applications are much larger and more complex than benchmarks previously used for automatic parallelization research [3].

The remainder of this work begins with Section 2, which discusses the analyses we have found that expose coarse-grained parallelism in media applications. Section 3 will discuss the forms of coarse-grained parallelism we target. Section 4 goes into detail on one version of the MPEG-4 encoder, which had the richest parallelism among the applications. Section 5 will discuss previous efforts on these analyses with respect to parallelization, and Section 6 concludes with a summary of our contributions.

2 Analyses

A compiler uses its analyses to refine its picture of a program. For the purpose of discovering parallelism, data dependence is one of the most important characteristics of the program. Analyses clarify the picture either by finding precise data dependences or by removing spurious ones. Because analyses must be conservative when supporting optimizations and transformations, the compiler's picture of the program is often cluttered by spurious dependences. For example, accesses to two memory objects are marked as conflicting if the objects cannot be proved independent. A single spurious dependence can prevent multiple opportunities for parallel execution.

Many different analyses with different aspects or levels of sensitivity have been derived to remove these spurious dependences in different situations. In order for the compiler to recognize different forms of parallelism, such as those

listed in Section 3, many different analyses must be combined or integrated. If critical analyses or specific options of an analysis are missing, existing parallelism will remain hidden.

The chief obstacle to discovering opportunities to parallelize a media application written in C/C++ is identifying dependences between pointer references (including references to arrays). A high-quality pointer analysis is essential in determining the relationship between pointer references. However, there are many coding constructs and programming practices that veil the true picture of memory usage from pointer analysis. For some of these cases, like recursive data structures and arrays, more specialized analyses such as shape analysis [8,32] and array analysis [9] can be very helpful in clarifying the picture.

In order to perform parallelization at the scale we are proposing, the analyses need to be scalable. The analyses can not be limited to only the parts of the program that are potentially parallelizable because those parts are coupled to the rest of the program by both pointer relationships and numeric values. Programs frequently possess diverse behaviors that are based upon input data, command-line flags, or defined constants. These settings are typically determined at the beginning of the program and propagate throughout the program code, and must be taken into account by the analyses.

It is certainly more expedient to rewrite a program to suit existing parallelizing analyses than to create new analyses sophisticated enough to understand existing programs. Nonetheless, there are practical benefits to be expected from an automated analysis framework. Programs may comprise many thousands or millions of lines of code and have multiple maintainers. Manually understanding and rewriting them is a tedious task that could benefit from automation. As programs become bigger and more complex, manual manipulation becomes even more difficult. More fundamentally, a more powerful set of analyses can be understood as granting the programmer more freedom to write flexible and modular code. For example, the IJG JPEG library stores all data related to the processing of a particular image in a dynamically allocated data structure. While many analyses would be more precise if the data were stored in global variables, to rewrite the library in such a manner would mean giving up reentrancy, which was designed into the library to make it usable in larger programs [18].

2.1 Pointer Analysis

Pointer analysis has been a subject of much research. At its core, pointer analysis determines what objects a memory reference can possibly access. The many extensions, like context-sensitivity, flow-sensitivity, and field-sensitivity, further specialize and clarify the picture pointer analysis presents by eliminating spurious dependences. Figuring out data dependence and data flow is very important for all of the different forms of parallelism in Section 3, and pointer analysis is essential to generate an accurate picture of the usage of memory objects. We will discuss the various features of pointer analysis we found indispensible for discovering parallelism. Implementing all of them in a single framework was a challenging task.

Pointer analysis needs to be scalable as well as precise. An important observation is that memory object allocation code and pointer assignments are often far from the usage of the objects, both in code location and execution time. Memory objects are frequently initialized towards the beginning of execution and used throughout the rest of the application. Of the two major options for pointer analysis, Andersen's-style (inclusion-based) and Steensgaard's-style (unification-based), Steensgaard's is generally cheaper and more scalable since it restrains the analysis' working set by merging the objects to which a pointer can reference. However, this can result in spurious dependences when any pointer, including those not in the parallel code region, can point to multiple objects. For this work, we used a scalable Andersen's-style analysis called FULCRA [26], which supports all options discussed in this section.

There are two pointer analysis options which have a highly synergistic effect and are generally necessary for adequate resolution of memory usage. First, the allocation function for a particular type of dynamically-allocated memory object is frequently reused to allocate multiple objects. A consequence of this code reuse is a need for the ability to distinguish certain objects that share a static allocation site, which we call *heap-sensitivity*. We specifically use heap specialization [27] enabled by a context-sensitive analysis to achieve heap-sensitivity. Context-sensitivity obtains a higher resolution in a pointer analysis by summarizing the pointer effects of procedures into their direct and indirect callers and obtaining information specific to the individual calling contexts. Heap specialization builds on top of this by versioning/cloning each heap object when pointer summaries are propagated to callers.

The other major analysis option is field sensitivity, which is needed because a non-field-sensitive pointer analysis will group together all of the objects pointed to by a structure. This prevents the compiler from distinguishing objects through those pointers. This case appears regularly since media programs commonly manipulate multiple data channels, and programmers use structures to organize data hierarchically. A natural way to organize channels of a single data set is to aggregate them as different fields of a larger structure.

As an example of the need for the multiple features of pointer analysis, consider the sample code in Figure 1. Without heap and context-sensitivity, the objects will be determined to be the same due to the similar calls to `AllocateBuffer`. Field-sensitivity is needed to distinguish `inphase` and `quadrature` as different fields of `signal`. Consequently field-sensitivity, context-sensitivity, and heap-sensitivity are all needed for the compiler to determine that the objects pointed to by the fields `inphase` and `quadrature` are independent.

The final pointer analysis option discussed here is flow-sensitivity. The default mode of operation for many pointer analyses is to not take into account the execution order of pointer assignments in programs, conflating the objects that the pointer references at different times. Flow-sensitivity instead includes ordering information into the analysis, but often at great cost to analysis working data size and runtime. The most common case where flow-sensitivity is useful is when a pointer is reused for different purposes or data at different program

locations. Several pointer analyses in the past have attempted to make a tradeoff between flow-sensitive and non-flow-sensitive analysis, often using a derivative of the SSA form [13]. Partial flow-sensitivity of this nature is also supported in FULCRA [29].

```
/* AllocateBuffer() calls calloc() to allocate memory. */
signal->inphase = AllocateBuffer (length);
singal->quadrature = AllocateBuffer (length);
```

Fig. 1. Code example that requires numerous pointer analyses

2.2 Array Analysis

When two pointers are known to reference the same object, array analysis can indicate whether or not the pointers reference the same memory location. This form of analysis conveys information about which loop iterations carry a dependence. Iterations are independent and can be executed in parallel if there are no flow, output, or anti-dependences between them. Array analysis can also determine if different loops access disjoint subsets of a given object. Finally, array analysis can be used to derive the data correlation between iterations of separate loops.

The common situation for parallel loops in media applications is to read input data from a segment of an array or set of arrays, process the data, and write the output data to a segment of a separate set of arrays. Although pointer analysis will eliminate spurious flow and anti-dependences between the read and written memory objects, array analysis is necessary to eliminate the output dependences between stores of different iterations. There are also loops where the input and output array locations are the same for each iteration, due to reuse of data structures. These loops also require array analysis to eliminate flow and anti-dependences. To be scalable, a compiler should extract symbolic expressions and perform induction variable analysis [6] on a demand-driven basis.

One important aspect of media applications is that they often have a range of supported sample rates, sizes, or resolutions and use many symbolic variables in the interest of code reuse. When dimensions are known integers, array analyses only need to handle affine expressions, where at most one symbolic variable (the loop inductor) exists for each multiplicative term. Dimensions determined at runtime create non-affine expressions and variable loop bounds, which stymie many simple array disambiguation tests [3]. In these cases, value constraints can be obtained or computed to assist the array disambiguation analysis [12,21]. More on value constraints and relationships is discussed in Sections 2.3 and 2.4. We have incorporated the Omega Test [31] and an extension of the I-Test [24] that manipulates symbolic expressions into our analysis infrastructure.

An example containing non-affine array expressions and variable loop bounds is shown in Figure 2. Four writes to the array `large` are performed per inner loop iteration. A basic array analysis that only looks at affine expressions

```
for (j = 0; j < height - 1; j++) {
  for (i = 0; i < width - 1; i++) {
    item = small[j*width + i];
    large[2*j*width + 2*i] = item;
    large[2*j*width + 2*i + 1] = item;
    large[2*j*width + 2*(i+width)] = item;
    large[2*j*width + 2*(i+width) + 1] = item;
  }
}
```

Fig. 2. Non-affine array expressions in a variable-bounded loop

with constant loop bounds would be unable to address this code, since one address could be reproduced by numerous combinations of `j`, `width`, and `i`. More advanced analyses incorporate information about the value ranges of variables relative to constants and each other. In this case, all three variables are known to be non-negative and `i` is always less than `width-1`, due to inner loop iteration conditions, which removes spurious output dependences and identifies the loops as having parallel iterations.

Because of modularity and code reuse, it is not uncommon to have procedure calls in a loop that operate on a different segment of an array every iteration. We would like to preserve access summaries for procedure calls for the appropriate contexts, since inlining can greatly increase the code size of an application. Without summary information, the compiler must assume that the procedure can access all elements of an array, which prevents parallelism if elements are written to in the array. Prior work exists for the Fortran language [28], but efficient summaries for the C language are still under development.

2.3 Value Constraint Analysis

Many variables in a program have a relatively small set of values during the majority of program execution, restricted by control flow tests or written constants. Information about the possible range or other constraints on their values can be critical in evaluating symbolic tests, such as the array analyses mentioned in Section 2.2. Value constraint information can also eliminate "dead" error checks within loops that create early exits; multiple loop iterations cannot be run non-speculatively unless these checks are removed. Many of these checks serve to detect bugs during program development and cannot actually occur during error-free execution. The size of contemporary applications necessitates an interprocedural, demand-driven method for finding these value constraints.

As an example of the value constraint problem, consider the code in Figure 3. In the example, `image->bits` is a pointer to a linear array holding a two-dimensional greyscale image of dimensions `width` and `height`. The main processing phase in this simplified example consists of four nested loops. The body of the second loop generates one eight-by-eight block of data and writes it into the image. To determine whether parallelization of LOOP1, LOOP2, or

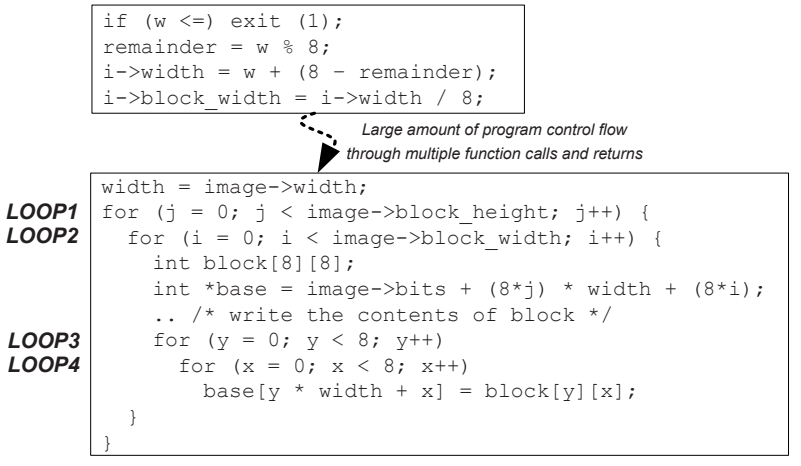


Fig. 3. Example of the value constraint and value relationship problems

LOOP3 is safe, an analysis needs to verify that there are no output dependences between the writes to `image->bits` in any iterations of those loops. If the absolute value of `width` is less than 8, then an output dependence exists between successive iterations of LOOP3, preventing independent execution of iterations of LOOP1, LOOP2, and LOOP3, as illustrated in Figure 4(a,b). If the compiler can locate the statement that generates the variable's value, it can determine that the alignment code restricts the value of `width` to a multiple of 8. It should be noted that contemporary applications often have this restricting code distant in code space and execution from the relevant uses, necessitating a whole-program analysis.

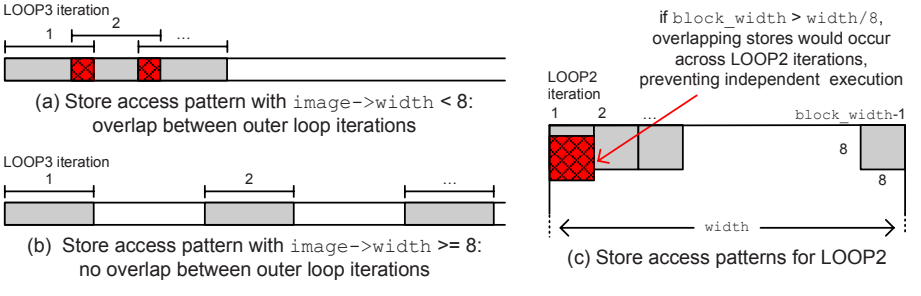


Fig. 4. Access patterns for the value constraint and relationship problems in Figure 3

Previous work on value constraint analysis has generally taken a dataflow approach to find valid ranges [12,21]. However, in contemporary C applications many values are loaded from dynamically-allocated memory, which require

more sophisticated methods to be effective. We are currently experimenting with demand-driven backtracking methods through memory objects to find constraint information.

2.4 Value Relationship Inference

In addition to knowing the set of values that a variable may retain, symbolic array disambiguation analyses gain precision from knowing the relationship between the values of different variables. Often, one variable is used to compute the value of several other variables. When related variables appear in an index expression, symbolic analyses typically lose precision unless they know the relationship between the variables. The compiler can find these relationships by tracking values back through def-use relationships to find common terms [35]. This requires interprocedural expression computation through memory objects, often dynamically-allocated, to find the relationships between values. As with the value constraint problem, an analysis for value relationships should be performed in a demand-driven manner to be scalable.

Analyses have been constructed to infer the relationship of values in the absence of dynamically-allocated memory [2], but in the applications we have studied the relevant values are passed via fields of heap structures and require more sophisticated memory analysis to track its definitions and usage. These analyses are currently under development.

Operating on one data set at multiple granularities is a common characteristic of media codes. For example, signal processing often divides a signal into segments containing a small number of samples, and image processing often divides an image into square blocks of pixels. In such applications, we have a common coding practice of precalculating the dimensions of the data set at each level of granularity during initialization. To see how this may confuse array disambiguation tests, consider again the code example in Figure 3. The relationship between the bound of LOOP2, `block_width`, and the variable `width` used in indexing the image array, is established when the image's memory is allocated, during the program's initialization phase. If during the execution of LOOP2 `i` reaches `width/8`, then there will be an output dependence across the outer two loops. Without knowledge of the relationship between the loop bound `block_width` and the variable `width`, a compiler must be conservative and not parallelize LOOP2 or LOOP1. However, if the compiler can trace back to the initialization code, it can determine that LOOP2's bound value is `width/8`, that `i` can be at most `width/8 - 1`, and that no output dependence exists.

Figure 4(c) shows the effects on the target image. Each iteration of LOOP2 fills in an 8x8 pixel block in the image. By default, the compiler must assume that the subset of the image that is written in the inner loop can "wrap around" and overwrite data written in an earlier iteration (the hashed block), introducing an output dependence that precludes parallelization. Value relationship information can tell the compiler that this access is not possible.

3 Forms of Parallelism

There are multiple forms of coarse-grained parallelism that can be exploited in media programs. We have divided them roughly into three different categories. The first is *loop-iteration parallelism*, or simply iteration parallelism, where different iterations of the same loop can be executed independently. The second is *region parallelism*, where separate static code regions can be executed independently. The final category is *cross-loop parallelism*, where results of one loop feed another, but not all iterations of the first loop need to be completed for the second loop to begin. This section will present the different types of parallelism in detail and the conditions that must be asserted before parallelism can be leveraged via transformations [22]. In order for the compiler to automatically detect and check the the necessary conditions for parallelism, the compiler will need to leverage all of the analyses outlined in Section 2.

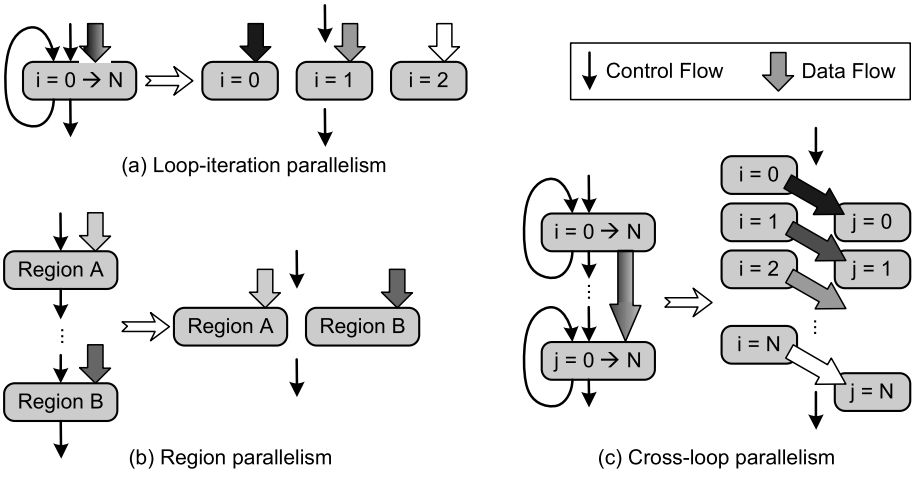


Fig. 5. Forms of loop parallelism

When discussing compiler-detected coarse-grained parallelism, the first concept that comes to mind is often loop-iteration parallelism, as shown in Figure 5(a). When each iteration of a loop depends on different data, as indicated by the different colors of data flow, the iterations can be separated and executed in parallel via loop distribution. Much of the previous work in this area has been performed in the Fortran programming language. The move to C/C++ introduces new issues, some of which are covered in [22].

Before exploiting loop-iteration parallelism, the compiler must assert that the iterations are truly independent. The memory locations written by each iteration must be independent from those of other iterations, and no iteration can write to another iteration's input data. This assertion quickly becomes complicated due to the usage of pointers and dynamically-allocated memory objects in C/C++.

Large-scope or whole-program capable analyses are also needed: benefit from small inner loops is limited and the allocation of memory objects may be distant in execution and code. Finally, sophisticated pointer analysis features may be needed to successfully disambiguate memory objects.

Certain regions of static code can always be executed independently if they do not have any data dependences between them. Figure 5(b) illustrates this case of parallelism, which we term region parallelism. Although this form of parallelism may appear easier to determine than loop-iteration parallelism, the same assertions regarding data independence must be made. Furthermore, the determination of region boundaries when trying to detect this form of parallelism is not a trivial task [34]. For software engineering reasons, it may well be the case that function boundaries also define reasonable task boundaries, but this may not provide optimal parallelism in all cases.

Cross-loop parallelism arises when one loop produces data that is consumed by a following loop. If each iteration of the following loop only depends upon a limited and known number of iterations of the previous loop and does not overwrite the first loop's input data, it is possible to execute part of the two loops in parallel as long as the real data dependences are respected. This parallelism is similar to that exploited in vector chaining, except at a much coarser granularity. Figure 5(c) shows an example of cross-loop parallelism, where each iteration of the second loop depends upon only the same iteration number of the first loop. The detection of cross-loop parallelism involves finding opportunities despite the presence of real dependences, rather than the absence of any dependence (as is the case for the previous two forms of parallelism). This necessitates the use of analyses to examine the data production/consumption patterns of different loops. Although this may be relatively trivial for some examples, more complicated situations exist that deal with data at different granularities and traversal order.

4 MPEG-4 Encoder Evaluation

To provide a greater understanding of the relative importance of various analyses, particularly in combination, we show the effects of these analyses on the degree of compiler-visible parallelism on an implementation of the MPEG-4 video encoder source code from the MPEG Industry Forum [25]. This implementation is specialized for MPEG-4 Simple Profile and optimized for execution on superscalar processors.

The size of the application is approximately 18 000 lines of code after removing comments and blank lines. There are 574 potentially called procedures with a total of 392 loops, and the maximum call depth is 10 procedures. Although the majority of execution time is spent in several dozen loops containing a few thousand lines of code, we emphasize that important and necessary information for analyses can be located anywhere in the application. Even so, our pointer analysis takes only 18 seconds to complete and uses 52 MB of memory on a 900 MHz Itanium 2 system. We will explain the basics of MPEG-4 encoding

here; for further information on the MPEG standard, readers are directed to documentation from the MPEG Industry Forum.

4.1 MPEG-4 Overview

An MPEG-4 Simple Profile stream consists of a series of images, called *frames*, that are processed one at a time. The two types of frames supported by Simple Profile are *I-frames* and *P-frames*. I-frames are encoded similarly to JPEG images and used as starting references for P-frames. A JPEG image consists of three component images: one full-resolution *luminance* image and two quarter-resolution *chrominance* images.[1] These are subdivided into *macroblocks*, consisting of six 8x8 pel ("picture element", effectively pixel) blocks. Four of these form a 16x16 pel square of the luminance image, and the other two are the corresponding chrominance blocks for the same part of the total image.

In P-frame encoding, the input image is reproduced as closely as possible by copying similar macroblocks from nearby locations in the previous I- or P-frame. The difference between the input and newly reconstructed is calculated and encoded as an image. Because the difference is usually very small, difference images are simple and highly compressible. P-frames thus take advantage of both spatial and temporal locality and are responsible for the majority of MPEG-4's compression. Consequently, typical encoding configurations use a high ratio of P-frames to I-frames. P-frame encoding dominates the execution time of the MPEG-4 encoder because of the increased processing required over that of I-frames and their prevalence in the video stream.

Figure 6 shows a dataflow diagram of the processing that occurs on each macroblock during P-frame encoding. First, *Motion Estimation* finds a macroblock in the previous frame that closely approximates the current macroblock.[2] *Motion Compensation* reproduces the estimated image using vectors from motion estimation. *Frame Subtraction* calculates the difference/error between the input macroblock and the macroblock copied from the previous image. *Discrete Cosine Transform (DCT) and Quantization* perform a JPEG-style encoding on each 8x8 pel error block. *Dequantization and Inverse Discrete Cosine Transform (IDCT)* decode the encoded image for use as the reference image for the following P-frame. Finally, *Bitstream Encoding* performs variable-length encoding to produce the final video bitstream. Rate control, which adds additional data dependences between these stages, and the option for printing motion estimation debug information have been disabled.

The main point of interest in Figure 6 is that only a single set of data dependences exists between macroblocks, consisting of flow dependences within

[1] Rather than encode in the three colors red, green, and blue, luminance and chrominance are used because the human eye is more sensitive to changes in luminance than chrominance. The standard takes advantage of this by having a smaller/coarser resolution for the chrominance component images.

[2] There are a wide range of motion estimation algorithms published; we chose a version for which there are no data dependences between motion estimation computation for different macroblocks.

bitstream encoding. Otherwise, the processing of different macroblocks can be executed in parallel. Certain parts of the computation can be further subdivided into separate luminance and chrominance components. We have parallelized this code by hand into multi-threaded implementations to confirm the parallelism.

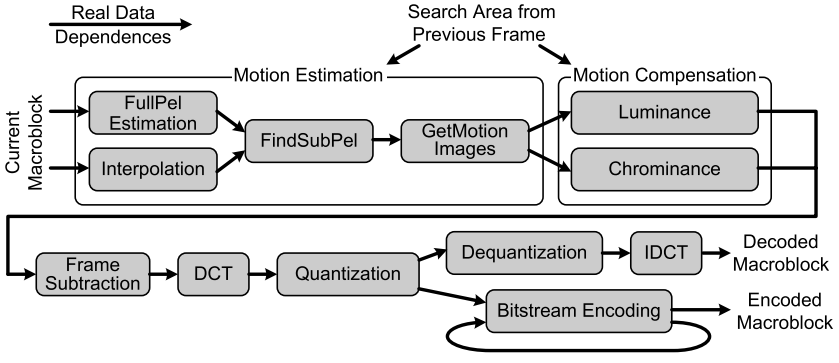


Fig. 6. A dataflow diagram of the processing of a single macroblock in the MPEG-4 encoder

Rather than perform all operations on a single macroblock at a time, as implied by Figure 6, the program performs subsets of the operations on the entire frame. It does not necessarily operate on a macroblock per loop iteration: for example, `FrameSubtraction` operates on single pixels. The pervasive use of pointers, differing traversal patterns, heap-allocated structures, and whole-program data and value flow make it difficult for the compiler to obtain a view of the application resembling Figure 6.

4.2 MPEG Analyses

The interaction and benefits of different analyses can be difficult to quantify in general, since individual analyses may fail to detect parallelism in isolation. Additionally, traditional metrics, such as points-to-sets, pairwise counts of independent memory operations, or weighted counts of parallelizable loops hold little meaning, as they may not indicate parallelization opportunity or represent parallelism of appropriate, practical granularity.

Because the purpose of this work is to identify parallelism rather than create a specific implementation, instead of a numeric metric we use a *loop-nest diagram* to express the parallelism visible to the compiler at different granularities. The specific example in Figure 7 is P-frame encoding. Each block in Figure 7 represents a loop or nested loops. Loop nesting is represented by blocks within blocks. The shading scheme represents the degree of iteration parallelism that the compiler can detect for each loop scope: black blocks cannot be safely parallelized without speculation, gray blocks can be parallelized if stores are ordered/serialized, and white blocks indicate completely iteration-parallel loops.

It is important to note that although the *iterations* of certain loops are not parallel, different *instances* of those loops may be parallel at a coarser granularity, such as an enclosing loop. The primary example of this are the loops within `MotionEstimatePicture`. Region parallelism is represented by blocks that are horizontally adjacent; horizontal lines passing through those blocks indicate spurious dependences that prevent safe parallel execution of those regions. For this example, cross-loop parallelism exists between every vertically neighboring pair of loops that exist at the same scope. This is often concealed by insufficient information about dependences or access patterns; the *blocked* opportunities are represented by an X at the shared boundary of the neighboring loops.

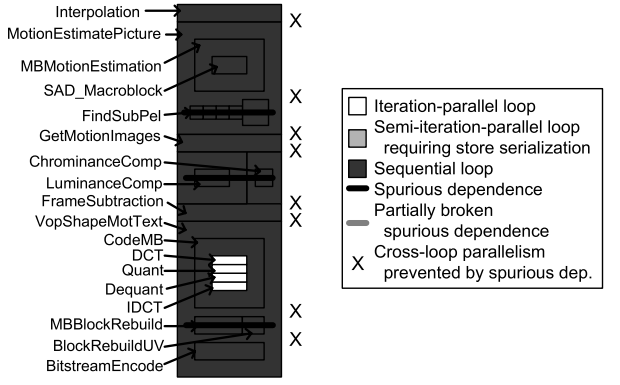
Only a handful of major loops are unparallelizable without algorithm changes in this application. `SAD_Macroblock` has an early exit in the loop, and is quite small in any case: 16 iterations with a maximum of approximately 100 instructions per iteration. `MBMotionEstimation`, `FindSubPel`, and `BitstreamEncode` have loops with data feedback. The non-parallel loops within `SAD_Macroblock`, `MBMotionEstimation`, and `FindSubPel` emphasize the importance of searching for parallelism at many loop levels. A system that only discovers inner-loop parallelism cannot parallelize the motion estimation code, one of the most compute-intensive regions of P-frame encoding execution, whereas Figure 7(i) shows that the loop-iteration parallelism of `MotionEstimatePicture` can be exposed to run multiple instances of `MBMotionEstimation` simultaneously.

Figure 7(a) shows the parallelism visible to the compiler with a context-insensitive, field-insensitive pointer analysis and array disambiguation analyses without non-affine expression or interprocedural support. The only loops that the compiler can identify as parallelizable are the small-granularity loops towards the bottom of the diagram. These loops are at most one to two hundred instructions per iteration, often much less, and are not well-suited for coarse-grained parallel execution. Figure 7(i) shows the parallelism that can be discovered if all of the analyses in Section 2 are used.

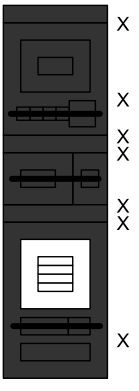
Figure 7(b,c,d,e) show the effects of adding a single feature to the analyses in (a), displaying the inability of isolated analysis features to expose the parallelism available in the application. Only a handful of opportunities are discovered, and these loops generally profit the least from parallelizing transformations since they do little data processing.

Figure 7(f,g,h,i) show one possible progression of combining analyses that expose more parallelism to the compiler (more white boxes and fewer lines and Xs). This does not imply that analyses should be run in this order: for example, all pointer analysis features would probably be run simultaneously, while interprocedural array disambiguation would be necessary only when a procedure boundary is encountered. For this application, the results of analyses do not exchange information and thus the ordering has no effect. The ordering was chosen solely to express the relative importance of particular analyses and options. The following subsections will discuss the effects of various analyses in greater detail.

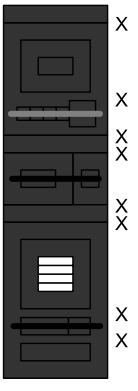
Although our pointer analysis framework supports both inclusion-based pointer analysis and partial-flow sensitivity, this application does not manifest situations



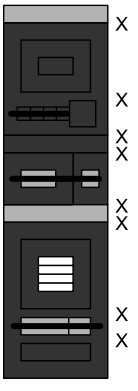
(a) **Original**: Affine expression intraprocedural array disambiguation analysis with a context-insensitive, field-insensitive pointer analysis



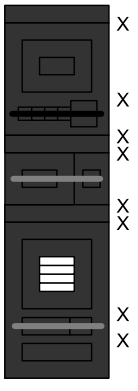
(b) Original + interprocedural array disambiguation



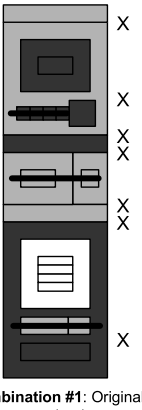
(c) Original + non-affine expression array disambiguation



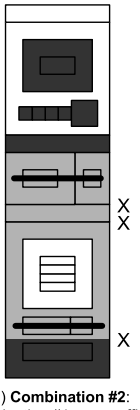
(d) Original + heap-sensitive, context-sensitive pointer analysis



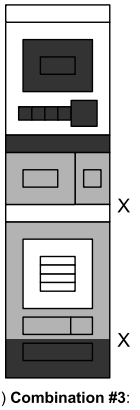
(e) Original + field-sensitive pointer analysis



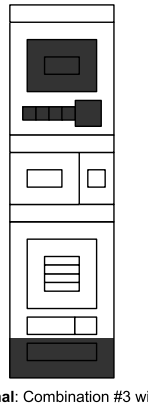
(f) **Combination #1**: Original + interprocedural array disambiguation + context- & heap-sensitive pointer analysis



(g) **Combination #2**: Combination #1 + non-affine expression array disambiguation



(h) **Combination #3**: Combination #2 + field-sensitive pointer analysis



(i) **Final**: Combination #3 with value constraint and relationship inference analyses

Fig. 7. Effects of various analyses on the compiler-visible parallelism in MPEG-4 P-frame encoding

where they add an appreciable benefit. This is due to the specialization for Simple Profile; the version of the reference MPEG-4 encoder that supports more features requires these analyses to identify important parallelism. As stated in Section 2, interprocedural array analysis and value constraints and relationships were computed by hand for this application, but all other analyses were performed automatically by the framework.

4.3 MPEG Loop Iteration Parallelism

Of the single analyses added to the original configuration in Figure 7(a), only interprocedural array disambiguation in Figure 7(b) exposes any reasonably coarse-grained iteration parallelism. This case occurs because the arrays being operated on are for a single macroblock, so they are of fixed dimension and statically allocated. Heap specialization in Figure 7(d) allows the parallelization of `Interpolation` and `FrameSubtraction` by distinguishing the input and output memory objects. However, these are very small loops and occupy less than 8% of the total execution time on a uniprocessor. Since the exposed parallelism is of a form that requires store serialization (ordering), it would likely be impractical to execute these loops' iterations independently.

Combining analyses exposes far more parallel loops to the compiler. In Figure 7(f), an interprocedural array disambiguation and heap-sensitive pointer analysis is able to discover that the loops that traverse over macroblocks in `MotionEstimatePicture` and motion compensation are iteration-parallel, despite the fact that loops within each iteration are not. Because these loops contain procedure calls that operate on different segments of an array each iteration, interprocedural array analysis is necessary to determine that the calls do not conflict. `MotionEstimatePicture` is a significant part of P-frame encoding, usually over 20% of total execution time on a uniprocessor with inexpensive, parallel algorithms. In addition, the output of each loop iteration is only four two-dimensional vectors and an integer indicating the encoding mode, so the store serialization requirement is not very significant.

The addition of non-affine expression array disambiguation in Figure 7(g) enables loop iterations in `MotionEstimatePicture` to execute in parallel without store serialization: it can determine that each iteration's output is to a different element of a variable-length array. In addition, it allows iterations of the upper portion of `VopShapeMotText` to be run in parallel with store serialization if they are split from the lower portion of the loop. This means that every macroblock can execute the `DCT` through `IDCT` sequence independently, as long as the writes back to the output frame object are serialized. This is another significant section of P-frame encoding, and can take over 50% of total uniprocessor execution time. Most of this time (70%) is due to `Quant`, which performs several division operations in each loop iteration.

Figure 7(h) uses field-sensitivity in combination with heap-sensitivity to distinguish different luminance and chrominance images when their references are stored into fields of a single structure. This allows complete parallelization of `FrameSubtraction`, although it is not a significant part of execution time. The

combination is also necessary to fully parallelize the remaining gray loops which become white in Figure 7(i).

Figure 7(i) shows the results of value range and relationship inference, exposing the remaining parallelism in the program. `GetMotionImages` is made parallelizable by optimization: an early exit in the loop was removed by value range analysis proving that the exit cannot be taken. The motion compensation loops `LuminanceComp` and `ChrominanceComp` are fully iteration-independent because value range and relationship information allowed the parallelization of inner and outer loops. For similar reasons, iterations of `MBBlockRebuild` and `BlockRebuildUV` are also now identified as fully independent, which allows iterations of the upper portion of `VopShapeMotText` to be executed independently when split from the lower portion of the function.

4.4 MPEG Region Parallelism

The instances of region parallelism in the MPEG encoder involve operations on either separate, dynamically-allocated objects linked to a common structure, or separate regions of an array accessed by repeated calls to a function. Thus, single analysis options are incapable of removing the dependences that block region parallelism. Only in Figure 7(d) can the compiler separate one call to `FindSubPel` from the other four, as it writes to a different array.

The combination of non-affine expression array disambiguation and heap-sensitive pointer analysis enables the independent execution of the four "smaller" calls to `FindSubPel` in Figure 7(g). With the addition of field-sensitivity in Figure 7(h), the compiler can finally distinguish different luminance and chrominance images even when they are stored into fields of a single structure. This removes the two remaining horizontal black lines from the diagram and fully exposes the major instances of code region parallelism in the application.

4.5 MPEG Cross-Loop Parallelism

As shown by the absence of Xs in Figure 7(i), there is a high degree of cross-loop parallelism available in P-frame encoding. The two largest obstacles to discovering the producer-consumer relationships for cross-loop parallelism are the need to determine the independence of memory objects accessed by the loop and the ordering of processing (or lack thereof) allows an overlap in execution. The former issue is resolved by a full-featured pointer analysis. The latter is complicated by the use of procedures, non-affine expressions, and the fact that the granularities and order of processing can be different for different loops. For example, `FrameSubtraction` loops over pixels, while its neighboring loops operate on macroblocks. This does not translate into a direct ratio of iterations: macroblock processing will operate on an 16x16 block from the upper left corner of a luminance image, while pixel processing will go across the top row of pixels in the image before beginning the next row. Without significant transformation, `FrameSubtraction` would have to process 16 rows of data before `VopShapeMotText` could begin execution.

Little cross-loop parallelism is exposed by analysis until a heap-sensitive pointer analysis is combined with an interprocedural, non-affine expression array analysis, shown in Figure 7(g). The heap-sensitivity distinguishes the input and output data objects for many loops, while the full-featured array analysis permits a pattern analysis. Field-sensitivity in Figure 7(h) removes another blocked case by clarifying the multiple objects referenced in `FrameSubtraction`. The remaining blocked cases require value range and relationship inference to obtain clear data production and consumption pattern information.

5 Related Work

This paper covers a large range of compiler work, incorporating various compiler analysis techniques, the effects of integrating these techniques, the parallelism they expose for an automated parallelizing compiler, and further development and refinement of compiler techniques. This is in the same spirit as previous work by Hendren and Nicolau [15] and Ghiya et al. [7]. Our work differs in that our focus is on discovering available parallelism for consumption by multi-processing microprocessors. We currently limit ourselves to media applications, which are high in parallelism but are pointer-heavy serial implementations. In this section we address some of the major works that have been published in these areas.

Several existing analysis techniques form the foundation for our investigation. These include pointer analysis, induction variable recognition, symbolic scalar analysis, and array disambiguation. For an overview of previous work in pointer analysis, we refer to [16]. Another report by Hind [17] identifies, in a similar vein to our work, idioms and common patterns in C programs that can adversely affect the precision of different kinds of pointer analysis. The pointer analysis framework used for this work is described in detail in [26].

For an overview of several important array disambiguation techniques, we refer to [9]. The problem of disambiguating index expressions containing symbolic values has also been noted by Blume and Eigenmann, who propose the Range Test [3] to handle such expressions. The Access Region Test [28] also disambiguates symbolic expressions, with greater flexibility. While our choice of disambiguation tests is tuned to the problem domain, additional tests could easily be added.

A number of tools and techniques exist for symbolic numerical analysis. Existing value range propagation algorithms [12,21] work on an intraprocedural scope or on code without dynamic memory to infer the possible sets of values for some data, enabling refinement of control flow and index expressions. However, value range propagation has not been implemented in a scalable form that tracks value ranges both interprocedurally and through heap-allocated data structures for the C language. Value relationship inference has been implemented for program analysis and static error detection [2]. Again, to the best of our knowledge, no existing tools track values through heap-allocated data structures.

At present there is a shortage of work on coarse-grained parallelization of C programs. Several compiler projects, including Polaris [3], ParaScope [23], and

SUIF [10] have emphasized the interprocedural nature of parallelizing analysis and code transformation. These have primarily focused on Fortran programs. However, the disparity between C and Fortran semantics has resulted in automatic parallelization developed for Fortran not enjoying the same level of success when applied to C. Of these projects, only SUIF has published research on compilation of C programs. Their reported parallelization results are only for Fortran benchmarks [10,11].

There are other approaches to parallelization that are alternative or complementary to compile-time analysis. One alternative is to augment a programming language such as C to allow the programmer to communicate more information about pointer relationships to the compiler [14]. Another is to use run-time disambiguation or speculation when compile-time analysis fails [4,36]. Salamí and Valero [33] find that using a compiler-generated run-time disambiguation test to select between a parallel and non-parallel version of a loop produces speedups in multimedia applications comparable to the speedups garnered by interprocedural pointer disambiguation. While code versioning is useful as a fallback mechanism, it causes code growth due to multiple versioning and overhead due to runtime tests. Even in systems with run-time disambiguation support, success in static parallelization will increase the execution efficiency of applications.

6 Conclusions and Future Work

This paper discusses several forms of parallelism and studies the analyses required to expose them in media applications. We have shown the types of code sequences and practices that require the use of certain analysis options. Distinctions have been made between analyses that are already satisfactory for finding parallelism and those that currently do not have practical or scalable solutions.

Using a reference MPEG-4 encoder, we have shown the importance of combining analyses to obtain a much clearer view of the parallelism present in the application than when run individually. Our evaluation showed that an interprocedural, non-affine expression array analysis and a heap-sensitive pointer analysis are required to expose the majority of parallelism in the application. A field-sensitive pointer analysis and value range and relationship inference are necessary to find the remainder of the parallelism in the application.

For future work, we will continue work on developing practical and scalable analyses for those analyses that are not yet practical or scalable in the C language. Work is ongoing to apply the analyses to other applications. We are also developing software tools to assist programmers and developers in finding parallelism for use in their own designs.

Acknowledgments

We would like to thank Hillery Hunter at IBM and John Sias for their advice and feedback, and the reviewers for their comments. This work was partially supported by the MARCO Gigascale Systems Research Center (GSRC).

References

1. Advanced Micro Devices. AMD Athlon 64 X2 dual-core product data sheet, May 2005.
2. ASTRÉE Static Analyzer. http://www.astree.ens.fr/.
3. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. Technical Report 1375, University of Illinois at Urbana-Champaign, 1994.
4. Mark Byler, James R. B. Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple version loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 312–318, 1987.
5. Cell Project at IBM Research. http://www.research.ibm.com/cell/.
6. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.
7. R. Ghiya, D. M. Lavery, and D. C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Design and Implementation*, pages 47–58, 2001.
8. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in c. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.
9. Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13,14):1741–1783, 1999.
10. M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Transactions on Programming Languages and Systems*, 27:662–731, 2005.
11. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
12. W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, May 1977.
13. R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998.
14. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.
15. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed System*, 1(1):35–47, January 1990.
16. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.
17. M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. Technical Report RC21510, IBM T. J. Watson Research Center, March 1999.
18. Independent JPEG Group. coderules.doc. Text file in zipped archive, 1998. ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz.

19. Intel Network Processors. http://www.intel.com/design/network/products/npfamily/.
20. Intel Pentium D Processor. http://www.intel.com/products/processor/pentium_D/ index.htm.
21. H. Johnson. Data flow analysis for 'intractable' imbedded system software. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 109–117, 1986.
22. K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.
23. K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2:329–341, 1991.
24. X. Kong, D. Klappholz, and K. Psarris. The I-Test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Parallel Languages and Compilers*, 2(3):342–349, July 1991.
25. MPEG Industry Forum. http://www.mpegif.org/.
26. E. M. Nystrom. *FULCRA Pointer Analysis Framework*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.
27. E. M. Nystrom, H.-S. Kim, and W. W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, June 2004.
28. Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems*, 24(1):65–109, 2000.
29. James Player. An evaluation of low-overhead partial flow-sensitivity. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2005.
30. Power Mac G5. http://www.apple.com/powermac/.
31. W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing 1991*, pages 4–13, November 1991.
32. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the ACM Symposium on Programming Languages*, pages 16–31, January 1996.
33. E. Salamí and M. Valero. Dynamic memory interval test vs. interprocedural pointer analysis in multimedia applications. *ACM Transactions on Architecture and Code Optimization*, 2(2):199–219, 2005.
34. V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 17–26, June 1986.
35. Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, 1995.
36. M. Voss and R. Eigenmann. Dynamically adaptive parallel programs. In *Proceedings of the International Symposium on High Performance Computing*, pages 109–120, May 1999.