A Design Tool for High Performance Image Processing on Multicore Platforms

Jiahao Wu*, Timothy Blattner†, Walid Keyrouz†, Shuvra S. Bhattacharyya*‡
*University of Maryland, College Park, MD, USA
Email: {jiahao, ssb}@umd.edu
†National Institute of Standards and Technology, Gaithersburg, MD, USA
Email: {timothy.blattner, walid.keyrouz}@nist.gov
‡Tampere University of Technology, Finland

***Abstract*—Design and implementation of smart vision systems often involve the mapping of complex image processing algorithms into efficient, real-time implementations on multicore platforms. In this paper, we describe a novel design tool that is developed to address this important challenge. A key component of the tool is a new approach to hierarchical dataflow scheduling that integrates a global scheduler and multiple local schedulers. The local schedulers are lightweight modules that work independently. The global scheduler interacts with the local schedulers to optimize overall memory usage and execution time. The proposed design tool is demonstrated through a case study involving an image stitching application for large scale microscopy images.**

I. Introduction

Making effective use of multicore platforms poses considerable challenges to designers of smart vision systems. These challenges include delivering real-time embedded computer vision performance under complicated data dependencies among application tasks, and the nondeterministic execution characteristics of thread-based programming models. Additionally, limited memory availability on embedded platforms along with the data intensive nature of smart vision applications necessitate special attention to memory management.

The Hybrid Task Graph Scheduler (HTGS) is a recently introduced application programming interface (API) and runtime system. HTGS significantly reduces the complexity of implementing scalable and high-performance multicore applications through a task-graph based application representation [1], and novel provisions for integrating optimizations for high performance multicore processing and memory management within this representation. Using HTGS effectively requires iterative performance tuning, which in turn requires significant designer effort to derive efficient schedules and manage memory usage patterns.

To automate the powerful but complex steps in the HTGS framework, we have developed the first version of a companion tool to HTGS in our recent work [2]. We refer to this companion tool as the HTGS Model-Based Engine (HMBE). HMBE further raises the abstraction level of the graphical representations in HTGS-based workflows so that the graphs are formulated in terms of dataflow principles. HMBE thereby enables dataflow-based analysis and optimization techniques to be integrated with implementations derived using HTGS. The class of analysis and optimization techniques enabled by HMBE include those associated with the *scheduling* of tasks and communication operations, which is a critical aspect of dataflow-based design processes for signal and information processing [3].

The HMBE scheduler provides lock-free and race-condition-free exploitation of parallelism without requiring programmer experience in parallel programming. HMBE integrates a modified form of ready list scheduling with an expanded representation of the input dataflow model that exposes parallelism effectively. The process of producing a multithreaded executable from the given input dataflow model is fully automated through the scheduling and code generation features incorporated in HMBE. In our experiments, we showed that when augmented with HMBE, HTGS yields comparable performance as HTGS without HMBE [2]. HMBE achieves this comparable level of performance while greatly reducing programmer effort through automation of scheduling and memory management processes.

In our previous work, HMBE was developed as a separate tool that links dataflow based modeling and scheduling with the software component (actor) library of HTGS. In this paper, we present a new version of HTGS in which HMBE is integrated seamlessly within HTGS rather than applied as a separate companion tool. We demonstrate that this new version provides significant further advantages compared to the previous, loosely-coupled integration between HMBE and HTGS. To distinguish it from the original HTGS tool, we refer to the new integrated toolset as HMBE-Integrated-HTGS (HI-HTGS).

HI-HTGS automates major parts of the HTGS-based design process using the model-based design methodology from HMBE, along with new developments that integrate HMBE tightly as a subsystem within HTGS. Advances involved in our development of HI-HTGS include (a) resolving the semantic mismatch between the schedulers of HTGS and HMBE through a hierarchical scheduling approach in which a global scheduler interacts with multiple local schedulers to automatically optimize overall memory usage and execution time; (b) replacing the application modeling interface with the Dataflow Interchange Format (DIF) Language [4]; and (c) migrating most of the setup-time and run-time analysis to compile-time, which results in a significantly more efficient,

978-3-9819263-0-9/DATE18/©2018 EDAA

lightweight runtime system.

We demonstrate the effectiveness of HI-HTGS through a case study involving an image stitching application for large scale microscopy.

II. Related Work

Many high-level languages and tools have been developed for signal processing applications that incorporate dataflow models of computation. Examples include CAL/Orcc [5], [6], PREESM [7], Multi-Dataflow Composer (MDC) tool [8], DIF [4], and HTGS [1]. HI-HTGS places a special emphasis on high performance multicore implementation, and integrated optimization of memory management and task scheduling using a single actor, dynamic invocation (SADI) scheduling model [2].

HI-HTGS combines the abstract dataflow graph analysis features of DIF with the APIs of HTGS, which enable construction, integration and iterative optimization of high performance software components and task graphs. While DIF focuses on high level dataflow analysis in which the detailed functionality of individual graph components (actors) is abstracted, HTGS provides extensive infrastructure for creating fully functional, high performance task graph implementations. Thus, the features of DIF and HTGS are highly complementary, and their integration through HI-HTGS provides new capabilities for automated, model-based analysis, implementation, and optimization of multicore signal and information processing systems.

Like HMBE, HI-HTGS applies an adapted version of Keinert's windowed synchronous dataflow (WSDF) model of computation [9]. WSDF is well suited as a model of computation for smart vision system design due to its provisions for representing image processing functionality. In WSDF, dataflow tokens can be placed in correspondence with multidimensional windows of data. These windows are restricted in WSDF to have constant dimensions. The degree of overlap between the windows can be configured through the *sampling matrices* parameter.

We incorporate minor adaptations to WSDF in the model of computation that we use in HI-HTGS. First, since the sampling matrix is a diagonal matrix, its representation can be simplified in the form of a vector. We apply this simplification in the adapted form of WSDF that we apply. Also, our adapted form of WSDF eliminates use of the *effective token size* parameter; instead, we assume that the basic unit of token production is a single token (as in conventional forms of dataflow).

We employ WSDF semantics in HI-HTGS due to the orientation of HI-HTGS toward high performance image processing and other areas of multidimensional signal processing. However, the scheduling techniques presented in this paper can readily be adapted to other forms of dataflow, such as synchronous dataflow, cyclo-static dataflow, and multidimensional synchronous dataflow, from which a certain type of expanded dataflow representation, called an acyclic precedence expansion graph (APEG) (or simply "acyclic precedence graph") [10], can be derived. In Section III, we discuss the APEG representation in more detail.

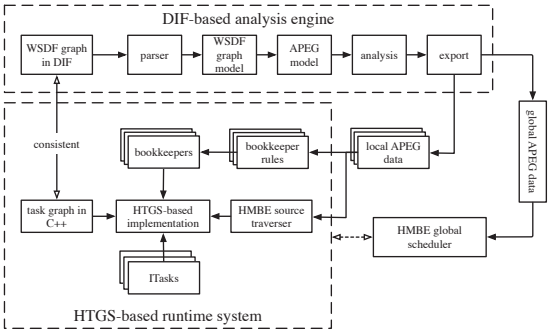


Fig. 1. Workflow associated with HI-HTGS.

In relation to the scheduling taxonomy of Lee and Ha [11], conventional scheduling techniques that apply the APEG and closely related representations are static or self-timed scheduling approaches [3]. In other words, these scheduling approaches fix the assignment of actors to processors at compile time as well as the ordering of actors that are assigned to the same processor. In contrast, HI-HTGS applies the APEG in conjunction with a more flexible scheduling model (the aforementioned SADI model), which is well-suited to operating on top of thread-based execution models that are employed with modern multicore processors. In particular, the SADI model enables run-time adaptivity of the schedule to cope with nondeterministic characteristics of thread-based execution, and dynamic variations in actor execution times. For more details on SADI scheduling, we refer the reader to [2].

III. Workflow

HI-HTGS is a design tool that enables high-performance image processing on multicore platforms. The tool is composed of two main parts: (a) the DIF-based analysis engine (*analysis engine*), which performs compile-time analysis for the application dataflow graph, and (b) the HTGS-based runtime system (*runtime system*), which applies the analysis engine, and provides optimized multithreaded execution and memory management for the application. Figure 1 illustrates the workflow associated with HI-HTGS.

To transfer dataflow graph analysis results from the analysis engine to the HTGS-based runtime system, and allow the HTGS-based runtime system to apply these results efficiently, we have developed a compact data exchange protocol using the Google Protobuf library [12]. Our protocol, called the *HI-HTGS exchange protocol*, ensures that the analysis engine and HTGS-based runtime system communicate reliably and efficiently even though they are developed in different languages (Java and C++, respectively). Due to the more compact binary format of the HI-HTGS exchange protocol compared to the XML or JSON formats, data can be exported and imported with lower runtime overhead, and with smaller storage cost. These features are important in HI-HTGS since a relatively large amount of data is exchanged between the two parts of the

tool, including the APEG representations that are constructed by the analysis engine.

As described in Section II, HI-HTGS incorporates an adapted form of WSDF as the system designer's entry point to the tool. The WSDF-based application model (*application graph*) is written in the DIF Language, as shown in the top left corner of Figure 1. Given a textual representation of the WSDF model in DIF, the analysis engine parses the representation and produces a graphical internal representation within the analysis engine. This internal representation is then expanded into an equivalent APEG representation using minor adaptations to WSDF of standard techniques for multirate to single rate synchronous dataflow (SDF) conversion [10].

To ensure functional correctness, the WSDF graph representation must be kept consistent with the C++ task graph representation that is developed by the designer in HTGS. This consistency requirement is represented in Figure 1 by the two arrows labeled "consistent". Consistency between the two parallel representations can be assessed as an integral part of the design process through component and subsystem-level testing that verifies functional equivalence. Well-known processes for automated execution of test suites can allow such testing to be integrated efficiently into the design process. Various authors have discussed this matter; we cite the work of Hamill as representing this body of work [13].

Intuitively, HTGS and DIF provide complementary programming formats for applications in HI-HTGS. Software components in HTGS, called *tasks*, specify detailed intra-task functionality that is integrated with mechanisms for high performance task execution. On the other hand, the WSDF actors specified in DIF expose formal properties associated with interactions among tasks. In general, the mapping between HTGS tasks and WSDF actors can be one-to-many, one-to-one, or many-to-one depending on the application and the designer's approach to model-based representation. The programming approach in HI-HTGS shifts programmer effort to that of task programming and model-based design (WSDF-based design in the current version of HI-HTGS), and alleviates the burden of inter-task scheduling and memory management. The result is a more productive and reliable form of design through the higher level of abstraction provided by the coupled task- and actor-based specification format.

The APEG represents the firings within a single iteration of a periodic schedule for the WSDF-based application graph. Each vertex v in the APEG has an associated actor $actor(v)$ in the application graph, and represents a distinct firing of $actor(v)$ within a periodic schedule. Thus, each actor in the WSDF graph is mapped to a set of one or more vertices in the APEG. The APEG is useful as an internal scheduling representation in part because it exposes parallelism and inter-actor communication at the level of application firings [3]. HI-HTGS uses only APEG information without reference to any specific periodic schedule. Instead, the schedule evolves dynamically under the control of the runtime system.

After construction of the APEG, the analysis engine performs various forms of analysis that are used to annotate the APEG vertices. These annotations are used by the runtime system when the application executes. After finishing its compile-time analysis on the APEG, the analysis engine attaches its analysis results as attributes of the APEG vertices, and exports the annotated APEG as binary data using the HI-HTGS exchange protocol. This binary data is then read by the runtime system at the beginning of execution when the application graph is launched.

HI-HTGS augments HTGS with a novel hierarchical scheduling approach to bridge the semantic mismatch between the original HTGS framework and the HMBE scheduler. In our context, the problem of bridging semantic mismatch is related in large part to the concept of task graph *bookkeepers* in HTGS [1]. Bookkeepers are task graph components that are specialized to maintain state. Bookkeepers are especially useful for efficiently managing communication among tasks that have complicated data dependency relationships.

While HTGS bookkeepers provide a modular abstraction for specifying powerful methods to optimize memory management and interprocessor communication, the design of bookkeepers is in general a complex task, and requires considerable expertise, and effort. The hierarchical scheduling approach developed within HI-HTGS includes features to automate the design, implementation, and integration of HTGS bookkeepers.

To enable more powerful global optimization in HI-HTGS, we have developed a hierarchical dataflow scheduling approach that integrates a global scheduler and multiple local schedulers. The HMBE global scheduler is used to maintain the global execution state of the underlying dataflow model. The HMBE global scheduler periodically retrieves information from the local schedulers, and uses this information to dynamically configure the execution priorities of the actors with the objective of optimizing thread contention.

In summary, HI-HTGS allows the system designer to work at a high level of abstraction, and automates challenging and time consuming tasks that are required to optimize use of HTGS. Additionally, the models and methods employed in the analysis engine for HI-HTGS are based on WSDF semantics and APEG representations, and can be adapted readily to other design processes and tools that are compatible with these models.

IV. Hierarchical Scheduling

In this section, we present the hierarchical scheduling approach that is employed in HI-HTGS. In this development, we refer to the input dataflow model as the *application graph* $G_m = (V_m, E_m)$, where V_m is the set of actors and E_m is the set of edges. We denote the APEG representation of G_m as $G_a = (V_a, E_a)$. As described in Section III, each vertex $v_a \in V_a$ is expanded from an actor $actor(v_a) \in V_m$ of the application graph. We refer to vertex $v_a \in V_a$ as an *invocation* of its associated application graph actor $actor(v_a)$. At compile time, the analysis engine automatically derives G_a from G_m, derives information for optimized scheduling from G_a, and

then exports this scheduling information along with the APEG to the runtime system using the HI-HTGS exchange protocol.

A. Local Schedulers for Bookkeepers

Hierarchical scheduling in HI-HTGS is designed to automate and systematically integrate two aspects of HTGS-based implementation that are especially time consuming and error prone. The first of these aspects is related to a specific type of HTGS task graph component called a *bookkeeper*. HTGS bookkeepers help to resolve data dependencies and coordinate buffer management between communicating tasks [1]. Each bookkeeper b is associated with a set $Q(b)$ of HTGS tasks that communicate with one another. A bookkeeper b is designed by specifying *rules* that coordinate the production and consumption of data by the tasks in $Q(b)$. While bookkeepers enable separation of concerns between efficient memory management and task implementation, design of bookkeepers is time consuming and requires significant programmer expertise.

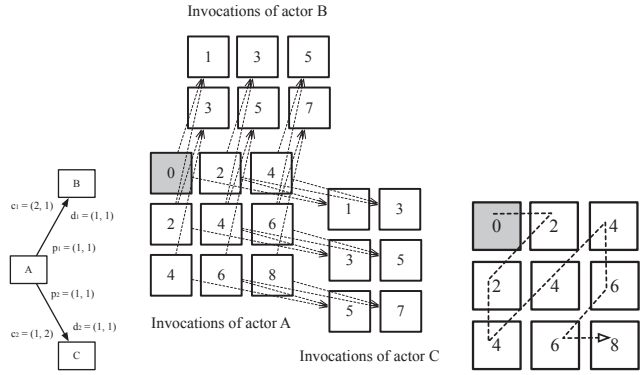
In HI-HTGS, the implementation of bookkeepers is automated through use of relevant information from the APEG. In particular, each bookkeeper is implemented in HI-HTGS by a *local scheduler*. An HI-HTGS local scheduler is a scheduling subsystem within the runtime system that uses local APEG information — that is, graph information that is related only to tasks in $Q(b)$. A local scheduler associated with bookkeeper implementation uses this graph information to dynamically manage data production and consumption, and optimize memory management associated with communication among tasks in $Q(b)$.

B. Local Schedulers for Source Actors

The second aspect of HTGS-based implementation that is automated in HI-HTGS through hierarchical scheduling is the process of determining how input data in multidimensional data streams is traversed to supply input to sequences of task executions. The traversal order can have a significant impact on performance. The enforcement of efficient traversal orders is performed automatically by local schedulers that are associated with source actors in the WSDF model (actors that model the interface between the system inputs and the task graph). These local schedulers are called *source traversers*. The source traversers developed in the current version of HI-HTGS assume that dataflow graph input is read from files.

In HI-HTGS, the problem of optimizing the traversal order for a source actor σ is modeled as the problem of determining the order in which the APEG vertices associated with σ are executed. This *APEG vertex execution order* (*AVEO*) is determined statically in HI-HTGS and enforced at run-time through information that is exported from the analysis engine to the runtime system.

The AVEO is computed using the undirected version U of the APEG — that is, the undirected graph that results from ignoring the direction of the edges in the APEG. The computation starts by deriving a starting vertex v_s in U that is associated with σ. The starting vertex is taken as an APEG vertex that has minimum out-degree from among all vertices



(a) Application graph. (b) APEG for the application graph. (c) Derived execution order for the source traverser.

Fig. 2. A simple example that illustrates derivation of the execution order for a source traverser in HI-HTGS.

associated with σ. If multiple vertices have minimum out-degree, then one of these is selected arbitrarily.

From the starting vertex, shortest paths in U are computed to all other vertices associated with σ. The path length used in this context is simply the number of vertices visited in a given path. After this process is complete, we have a path length $p(v)$ for every APEG vertex associated with σ. The vertices associated with σ are then ordered for execution by nondecreasing order of $p(v)$. Again, ties are broken arbitrarily. The local scheduler associated with σ enforces the $p(v)$-sorted ordering at run-time.

Intuitively, all steps of our AVEO derivation heuristic are designed to minimize the lifetime of data and improve data locality as the graph is executed, which are identical to the objectives of HTGS bookkeepers.

C. Source Traverser Example

Figure 2 illustrates with a simple example the derivation of the execution order for a source traverser. Figure 2(a) shows an example of a WSDF-based application graph that contains a source actor A. Here, each p_i represents the two-dimensional production rate associated with the ith edge, and similarly, each c_i represents the consumption rate. Each d_i specifies the movement of a sliding window on each edge from one invocation of the edge's sink actor to the next. For more details on the multidimensional production rates, consumption rates, and sliding window parameters associated with WSDF edges, we refer the reader to Keinert et al. [9].

In WSDF, the APEG in general depends on the dimensions of each frame of input data on which the application graph operates. Figure 2(b) shows the APEG that results from the application graph of Figure 2(a) with a frame size of 3×3 tokens. Here, squares represent APEG vertices and dashed arrows correspond to APEG edges. The vertices are laid out in the figure according to the two-dimensional index spaces associated with the corresponding WSDF actors [9]. For example, the APEG vertex in the second row and third column for actor A corresponds to invocation $(2, 3)$ of the

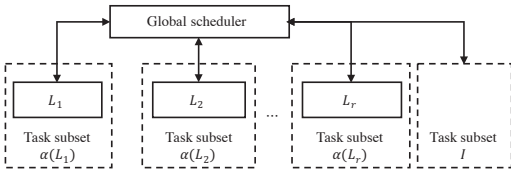


Fig. 3. An illustration of interactions among the global scheduler, local schedulers, and independent tasks in HI-HTGS.

actor. The number inside each square in Figure 2(b) gives the shortest path length between the corresponding APEG vertex and an invocation associated with the source actor A.

To determine the schedule for the source traverser associated with A, we sort the APEG vertices associated with A in nondecreasing order of their shortest path lengths. A schedule that results from such a sorting operation is illustrated in Figure 2(c). The dashed path illustrated in the figure shows the order in which invocations are dispatched starting with invocation $(1, 1)$ and ending with invocation $(3, 3)$.

D. Hierarchical Scheduling Architecture

In HI-HTGS, a given application graph G_m is scheduled using a set $\lambda = L_1, L_2, \ldots, L_r$ of local schedulers, which are in one-to-one correspondence with the bookkeepers in the associated HTGS graph and the source actors in G_m. The actors V_m in G_m are partitioned into $(r + 1)$ disjoint sets $\alpha(L_1), \alpha(L_2), \ldots \alpha(L_r), I$, where each $\alpha(L_i)$ represents the set of actors associated with local scheduler L_i, and I represents the (possibly empty) set of actors that are not associated with any local scheduler. In this context, we refer to I as the set of *independent actors*.

A global scheduler coordinates execution across the local schedulers along with their associated actors, and the independent actors. In our current implementation of HI-HTGS, we use a purely data-driven global scheduling strategy where actor invocations are dispatched for execution as soon as they have sufficient data and (if applicable) their local schedulers select them for execution. Investigation of more sophisticated global scheduling strategies in HI-HTGS is an interesting direction for future work.

Figure 3 illustrates interactions among the global scheduler, local schedulers, and independent tasks in HI-HTGS. Each local scheduler L_i and each source actor in G_m is assigned a separate thread in the targeted multicore architecture. Each actor that is not a source actor is assigned N_c threads, where N_c is the number of processor cores. The total number of threads for G_m can therefore be expressed as $r+N_s+N_c\times(N_m-N_s)$, where N_m and N_s represent the total number of actors in G_m and the number of source actors in G_m, respectively. During run-time, an actor invocation executes when it has been dispatched and one of the threads allocated to the actor is available to execute the invocation.

V. Case Study

In this section, we experimentally study the performance of HI-HTGS on an image stitching application for large-scale microscopy. In this application, a microscope collects a grid of overlapping images. Each image in the grid is called a *tile*. The objective of image stitching is to derive positional translations between adjacent pairs of image tiles, and integrate the translated tiles into a single mosaic.

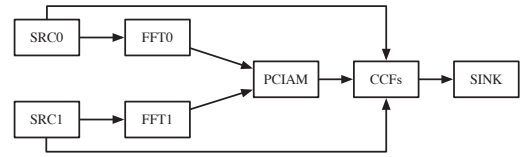


Fig. 4. HTGS task graph for the image stitching application.

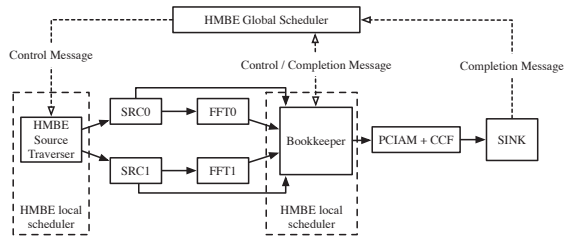


Fig. 5. WSDF model of the image stitching application.

The HTGS task graph that we use to implement the image stitching application is shown in Figure 4. The task graph iteratively loads adjacent pairs of tiles, computes their fast Fourier transforms (FFTs), applies the *phase correlation image alignment method* (*PCIAM*) [14] to the FFT results, and then extracts translation parameters from the results of the PCIAM operations.

This application is data intensive. Because each image tile is as large as 3 MB and its converted FFT result is about 12 MB, the application needs to allocate at least 15 MB of memory for a single image tile. The dataset that we experimented with is a 42×59 grid of 2478 image tiles. The grid encompasses over 35 GB of pixel data. If intermediate results are not released in a timely manner, the application can quickly run out of physical memory on the targeted computing platform.

To implement the image stitching application using HI-HTGS, we model the application as a WSDF graph and specify this graphical model using the DIF Language. Our WSDF model of the application is illustrated in Figure 5.

We experimented with the HTGS- and HI-HTGS-based implementations of the image stitching application on three different platforms, including a mid-range workstation, a low cost laptop computer and an outdated desktop computer. These platforms, summarized in Table I, are selected to help demonstrate the scalability of HI-HTGS, as well as the applicability of HI-HTGS to diverse targets.

Our experimental results involving execution time comparison are summarized in Table II. Here, we compare the measured execution times among our HTGS-, HMBE-, and HI-HTGS-based implementations of the image stitching application. The HMBE-based implementation is derived by automatically generating the APEG using a C++ extension of

TABLE I
THE PLATFORMS THAT WE USED IN OUR EXPERIMENTS.

Platform Name	CPU	Logical Cores	RAM	Disk Reading Speed (MB/s)
Mid-range Workstation	3.4 GHz Intel i7-2600K	8	16 GB DDR3	116.31
MacBook Pro A1502	2.7 GHz Intel i5-5257U	4	8 GB DDR3	1335.93
Outdated Desktop	2.8 GHz Intel Pentium D	2	2 GB DDR1	68.52

TABLE II
COMPARISON OF EXECUTION TIME.

Platform Name	HMBE	Sequential Implementation	HTGS	HI-HTGS
Mid-range Workstation	50.51 s	586.07 s	49.01 s	48.94 s
MacBook Pro A1502	108.57 s	256.75 s	104.06 s	103.86 s
Outdated Desktop	406.30 s	863.60 s	398.28 s	397.33 s

HTGS, and manually integrating the APEG with the HMBE scheduler, which is external to HTGS. Like the process of configuring bookkeepers in HTGS, this process of manually integrating the APEG is time consuming and error prone. Both of these tasks — bookkeeper configuration and APEG-to-scheduler integration are fully automated in HI-HTGS. Thus, HI-HTGS greatly simplifies and accelerates the process of deriving implementations that are equipped with the powerful performance optimization features of HTGS. From the results shown in Table II, we see that these improvements are delivered while maintaining performance levels that are similar to HMBE and HTGS.

To help assess the improvement in software development and maintenance cost provided by HI-HTGS, we compared the source code size of our HI-HTGS based implementation of image stitching with that of the original HTGS based implementation. We measured the source code size as the lines of code in all relevant source files. In these measurements, we considered all of the application-specific source code used for application setup, scheduling, and memory management — that is, all source code excluding code that is part of the general HTGS and HI-HTGS frameworks, and outside of the task (kernel) implementations, which can be reused across different applications. The results of these measurements are summarized in Table III, which shows a reduction in source code size of 63% achieved by HI-HTGS.

TABLE III
COMPARISON OF SOURCE CODE SIZE.

Tool	C++ code	DIF code	Total code	Improvement
HTGS	909	0	909	—
HI-HTGS	273	64	337	63%

VI. CONCLUSIONS

In this paper, we have presented a software tool for design and implementation of multicore image processing systems. This tool consists of two main parts — the DIF-based analysis engine, which applies the Dataflow Interchange Format (DIF) Package, and the HTGS-based runtime system, which builds on the Hybrid Task Graph Scheduler (HTGS). The tool allows system designers to incorporate powerful techniques for performance optimization and memory management while specifying applications at a high level of abstraction and using significant amounts of automation. Our experiments demonstrate the ability of our new design tool to provide this high level of abstraction and automation while generating efficient implementations on a diverse set of platforms. Useful directions for future work include extending the hierarchical scheduling techniques developed in this work to heterogeneous platforms, such as CPU/GPU platforms.

DISCLAIMER

No approval or endorsement of any commercial product by NIST is intended or implied. Certain commercial software, products, and systems are identified in this report to facilitate better understanding. Such identification does not imply recommendations or endorsement by NIST, nor does it imply that the software and products identified are necessarily the best available for the purpose.

REFERENCES

[1] T. Blattner, W. Keyrouz, M. Halem, M. Brady, and S. S. Bhattacharyya, "A hybrid task graph scheduler for high performance image processing workflows," in *Proceedings of the IEEE Global Conference on Signal and Information Processing*, 2015, pp. 634–637.

[2] J. Wu, T. Blattner, W. Keyrouz, and S. S. Bhattacharyya, "Model-based dynamic scheduling for multicore implementation of image processing systems," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2017.

[3] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009, iSBN:1420048015.

[4] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[5] J. Eker and J. W. Janneck, "Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 2012, pp. 1120–1124.

[6] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: multimedia development made easy," in *Proceedings of the ACM International Conference on Multimedia*, 2013, pp. 863–866.

[7] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan, "An open framework for rapid prototyping of signal processing applications," *EURASIP Journal on Embedded Systems*, vol. 2009, January 2009, article No. 11.

[8] F. Palumbo, N. Carta, and L. Raffo, "The multi-dataflow composer tool: A runtime reconfigurable HDL platform composer," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, 2011.

[9] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2006.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[11] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, vol. 2, 1989, pp. 1279–1283.

[12] Google, Inc., "Protocol buffers," 2017, https://developers.google.com/protocol-buffers/, visited on April 26, 2017.

[13] P. Hamill, *Unit Test Frameworks*. O'Reilly & Associates, Inc., 2004.

[14] C. D. Kuglin and D. C. Hines, "The phase correlation image alignment method," in *Proceedings of the International Conference on Cybernetics and Society*, 1975, pp. 163–165.