EDICIONES
Universidad Cooperativa de Colombia

Ingeniería Solidaria
ISSN (en línea): 2357-6014
Vol. 14, No. 25 [special issue], May 2018
Artículo de investigación / Research article
doi: https://doi.org/10.16925/.v14i0.2230

Static Worst-Case Execution Time Optimization using DPSO for ASIP Architecture

El peor caso estático de optimización del tiempo de ejecición utilizando DPSO para arquitectura ASIP

Mood Venkanna[1]✉, Rameshwar Rao[2]

[1] Osmania University, Hyderabad, India
[2] Osmania University, Hyderabad, India

✉ H.No:3-11-100,R T C Colony ,LB Nagar Hyderabad 500074,India. Email: principal@smec.ac.in; venkatmood03@gmail.com

Received on: September 30th, 2017 **Accepted on:** December 10th, 2017 **Available online:** May 1st, 2018

How to cite this article: M. Venkanna and R. Rao,“Static Worst-Case Execution Time Optimization using dpso for asip Architecture”, *Ing. Sol.*, vol. 14, no. 25 (Special issue), pp. 11, May 2018. doi: https://doi.org/10.16925/.v14i0.2230

Abstract

Introduction: The application of specific instructions significantly improves energy, performance, and code size of configurable processors. The design of these instructions is performed by the conversion of patterns related to application-specific operations into effective complex instructions. This research was presented at the ICITKM Conference, University of Delhi, India in 2017.

Methods: Static analysis was a prominent research method during late the 1980’s. However, end-to-end measurements consist of a standard approach in industrial settings. Both static analysis tools perform at a high-level in order to determine the program structure, which works on source code, or is executable in a disassembled binary. It is possible to work at a low-level if the real hardware timing information for the executable task has the desired features.

Results: We experimented, tested and evaluated using a H.264 encoder application that uses nine CIs, covering most of the computation intensive kernels. Multimedia applications are frequently subject to hard real time constraints in the field of computer vision. The H.264 encoder consists of complicated control flow with more number of decisions and nested loops. The parameters evaluated were different numbers of A partitions (300 slices on a Xilinx Virtex 7each), reconfiguration bandwidths, as well as relations of CPU frequency and fabric frequency f_{CPU}/f_{fabric}. f_{fabric} remains constant at 100MHz, and we selected a multiplicity of its values for f_{CPU} that resemble realistic units. Note that while we anticipate the WCET in seconds (WCETcycles/ f_{CPU}) to be lower (better) with higher f_{CPU}, the WCET cycles increase (at a constant f_{fabric}) because hardware CIs perform less computations on the reconfigurable fabric within one CPU cycle.

Conclusions: The method is similar to tree hybridization and path-based methods which are less precise, and to the global IPET method, which is more precise. Optimization is evaluated with the Discrete Particle Swarm Optimization (DPSO) algorithm for WCET. For several real-world applications involving embedded processors, the proposed technique develops improved instruction sets in comparison to native instruction sets.

Originality: For WCET estimation, flow analysis, low-level analysis and calculation phases of the program need to be considered. Flow analysis phase or the high-level of analysis helps to extract the program’s dynamic behavior that gives information on functions being called, number of loop iteration, dependencies among if-statements, etc. This is due to the fact that the analysis is unaware of the execution path corresponding to the longest execution time.

Limitations: This path is executed within a kernel iteration that relies upon the nature of MB, either I-MB or P-MB, determined by the motion estimation kernel, that is, its’ input depends on the I-MB and P-MB paths ,which also contain separate CIs leading to the instability of the worst-case path, that is, adding more partitions to the current worst-case path can result in the other path becoming the worst case. The pipeline stalls for the reconfiguration delay and continues when entering the kernel once the reconfiguration process finishes.

Keywords: embedded processor, specific integrated circuit application, worst case execution time, particle swarm optimization, discrete particle swarm optimization.

Research article

doi: https://doi.org/10.16925/.v14i0.2230

El peor caso estático de optimización del tiempo de ejecución utilizando DPSO para arquitectura ASIP

Resumen

Introducción: la aplicación de instrucciones específicas mejora significativamente la energía, el rendimiento y el tamaño del código de los procesadores configurables. El diseño de estas instrucciones se realiza mediante conversión de patrones relacionados con operaciones específicas de la aplicación con instrucciones complejas y efectivas. Esta investigación se presentó en la Conferencia ICITKM, Universidad de Delhi, India en 2017.

Métodos: el análisis estático fue un método de investigación prominente durante la década de 1980; sin embargo, las mediciones de extremo a extremo son un enfoque convencional en los entornos industriales. Ambas herramientas de análisis estático se desempeñan a un alto nivel para determinar la estructura del programa que funciona en el código fuente, o que se ejecuta en un binario desmontado. Es posible trabajar a bajo nivel si la información de tiempo de *hardware* real para la tarea ejecutable presenta las características deseadas.

Resultados: experimentamos, probamos y evaluamos con una aplicación de codificación H.264 que utiliza nueve elementos de configuración y cubre la mayoría de los núcleos de cálculo intensivo. Las aplicaciones multimedia están frecuentemente sujetas a duras restricciones en tiempo real en el campo de la visión por computador. El codificador H.264 consiste en un complicado flujo de control con más número de decisiones y bucles anidados. Los parámetros evaluados fueron de diferentes números de particiones A (300 cortes en un Xilinx Virtex 7 cada uno) y anchos de banda de reconfiguración, así como de relaciones de frecuencia de CPU y frecuencia de $_{fabric}$ fCPU/f_{fabric}. f_{fabric} permanece constante a 100MHz. Seleccionamos varios de sus valores para fCPU que se asemejan a unidades realistas. Es importante tener en cuenta que aun cuando anticipamos el WCET en segundos (ciclos WCET/ fCPU) para que fuesen inferiores (mejores) con fCPU más alta, los ciclos WCET aumentan (en un tejido constante f) porque los CI de *hardware* realizan menos cálculos en el tejido reconfigurable dentro de una CPU de ciclo.

Conclusiones: el método es similar a la hibridación de árboles y métodos basados en rutas, los cuales son menos precisos, y al método I pet global, que es más preciso. La optimización se evalúa con el algoritmo de optimización de enjambre de partículas discretas (DPSO) para WCET. Para varias aplicaciones del mundo real que involucran procesadores integrados, la técnica propuesta desarrolla conjuntos de instrucciones mejorados en comparación con los conjuntos de instrucciones nativas.

Originalidad: para la estimación de WCET se debe considerar el análisis de flujo, el análisis de bajo nivel y las fases de cálculo del programa. La fase de análisis de flujo o alto nivel de análisis, ayuda a extraer el comportamiento dinámico del programa que proporciona información sobre las funciones que se invocan, el número de iteraciones de bucle, las dependencias entre sentencias *if*, etc. Esto se debe a que el análisis desconoce la ruta de ejecución correspondiente al tiempo de ejecución más largo.

Limitaciones: esta ruta se ejecuta dentro de una iteración del núcleo que depende de la naturaleza de MB, ya sea I-MBO P-MB, determinada por el núcleo de estimación de movimiento, es decir que su entrada depende de las rutas I-MB y P-MB, que también contienen elementos de configuración separados que conducen a la inestabilidad de la ruta del peor de los casos, es decir que agregar más particiones a la ruta actual del peor de los casos puede hacer que la otra ruta se convierta en el peor de los casos. La tubería se detiene por la demora de reconfiguración y continúa al ingresar al núcleo una vez que finaliza el proceso de reconfiguración.

Palabras clave: procesador integrado, aplicación de circuitos específicos integrados, el peor tiempo de ejecución de casos, optimización por enjambre de partículas, optimización discreta por enjambre de partículas.

O pior caso estático de otimização do tempo de execução utilizando DPSO para arquitetura ASIP

Resumo

Introdução: a aplicação de instruções específicas melhora significativamente a energia, o desempenho e o tamanho do código dos processadores configuráveis. O desenho dessas instruções é realizado mediante a conversão de padrões relacionados com operações específicas da aplicação com instruções complexas e efetivas. Esta pesquisa foi apresentada na Conferência ICITKM, Universidade de Délhi, Índia em 2017.

Métodos: a análise estática foi um método de pesquisa proeminente durante a década de 1980; contudo, as medições de extremo a extremo são uma abordagem convencional nos contextos industriais. Ambas as ferramentas de análise estática se desempenham a um alto nível para determinar a estrutura do programa que funciona no código fonte ou que se executa num binário desmontado. É possível trabalhar a baixo nível se a informação de tempo de hardware real para a tarefa executável apresentar as características desejadas.

Resultados: experimentamos, testamos e avaliamos com uma aplicação de codificação H.264 que utiliza nove elementos de configuração e cobre a maioria dos núcleos de cálculo intensivo. As aplicações multimídias estão com frequência sujeitas a duras restrições em tempo real no campo da visão por computador. O codificador H.264 consiste num complicado fluxo de controle com mais número de decisões e circuitos aninhados. Os parâmetros avaliados foram de diferentes números de particiones A (300 cortes num Xilinx Virtex 7 cada um) e largos de banda de reconfiguração, bem como de relações de frequência de cpu e frequência de $_{fabric}$ fCPU/f_{fabric}. f_{fabric} permanece constante a 100MHz. Selecionamos vários de seus valores para fCPU que são semelhantes a unidades realistas. É importante considerar que, ainda quando antecipamos o WCET em segundos (ciclos WCET/ fCPU), para que fossem inferiores (melhores) com fCPU mais alta, os ciclos WCET aumentam (num tecido constante f) porque os CI de *hardware* realizam menos cálculos no tecido reconfigurável dentro de uma CPU de ciclo.

Conclusões: o método é similar à hibridação de árvores e métodos baseados em rotas, os quais são menos precisos, e ao método I pet global, que é mais preciso. A otimização é avaliada com o algoritmo de otimização por enxame de partículas discretas (DPSO) para WCET. Para várias aplicações do mundo real que envolvem processadores integrados, a técnica proposta desenvolve conjuntos de instruções melhoradas em comparação com os conjuntos de instruções nativas.

Originalidade: para a estimativa de WCET, deve-se considerar a análise de fluxo, a análise de baixo nível e as fases de cálculo do programa. A fase de análise de fluxo ou alto nível de análise ajuda a extrair o comportamento dinâmico do programa que proporciona informação sobre as funções invocadas, sobre o número de iterações de circuito, as dependências entre sentenças *if*, etc. Isso se deve a que a análise desconhece a rota de execução correspondente ao tempo de execução mais longo.

Limitações: essa rota é executada dentro de uma iteração do núcleo que depende da natureza de MB, seja I-MB, seja P-MB, determinada pelo núcleo de estimativa de movimento, quer dizer que sua entrada depende das rotas I-MB e P-MB, que também contêm elementos de configuração separados que conduzem à instabilidade da rota do pior dos casos; em outras palavras, adicionar mais partições à rota atual do pior dos casos pode fazer com que a outra rota se converta no pior dos casos. A tubulação se detém pela demora de reconfiguração e continua ao ingressar no núcleo assim que finaliza o processo de reconfiguração.

Palavras-chave: processador integrado, aplicação de circuitos específicos integrados, o pior tempo de execução de casos, otimização por enxame de partículas, otimização discreta por enxame de partículas.

1. Introduction

The application of specific processing elements needs modern optimized embedded systems. Application Specific Instruction Set Processors (ASIPS) are crucial for the desired physical and functional constraints of an embedded system. These must maintain high programmability and flexibility.

For a particular application domain, performance and power optimization of the processing elements are essential. The optimizations must include vector processing, complex domain-specific arithmetic operations, SIMD support, etc., providing extended instruction sets to the processor. The architecture organization comprises register files with specific configurations (data width, depth, or port size), local memories of application data, real-time data flow customized channels, and synchronization ports with respect to various SOC blocks. Appropriate emulation techniques are desired to provide optimized configuration possibilities by exploring customization of software_hardware systems for accuracy and performance estimates. For this purpose, the classical hardware characterization and functional metric applications such as the execution time, resource congestion and cache performance need to be optimized. Early physical metrics estimations that include the occupied area and energy/ power consumption are also necessary. Thus, hardware-based emulation techniques are an alternative to such problems, which require a scalable and accurate software-based simulation approach.

Stringent time constraints are essential to any hard real-time system and can be derived from the corresponding system under control. These constraints need to satisfy the upper limits of the respective execution times. However, in general it is not easy to maintain this for any program, hence the halting problems remain difficult to solve. Real-time systems however, have programming restrictions that guarantee the culmination of a program since recursions are never allowed and are bounded to the chosen iteration loop count.

The work assumes a real-time system with a number of tasks that can guarantee the desired functionality. Fig. 1 provides the relevant properties corresponding to a real-time task. Each task demands a specific execution time variation based on either environmental behavior of input data. The upper curve indicates the execution times in which best-case execution time (BCET) and worst-case execution time (WCET) denotes the shortest and the longest execution times respectively. Mostly the state space is very large to explore every possible execution exhaustively to find out the exact BCET as well as the WCET. In one approach, the execution time measurement considers a subset of all possible executions in order to compute the minimal and maximal observed execution times. In general, the approach overestimates the BCET and underestimates the WCET. Nowadays a common approach in many industries is to determine the execution-time bounds and it is known as the dynamic timing analysis.

The WCET analysis provides a priori information on a program's worst possible execution time before the program is used in any system. Reliable WCET estimation is essential in real-time systems particularly, when the systems control the safety and critical segmentations like military equipment, vehicles, and power plants.

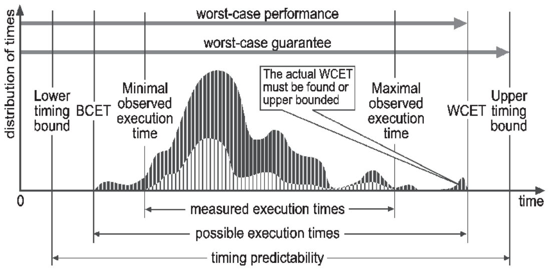

Fig . 1. Real-time system with wcet problem
Reference: [1]-[12].

The lower curve corresponds to a subset of measured executions. The darker curve, an envelope of the former, represents the times of all executions. It's minimum and maximum are the best-case and worst-case execution times respectively, abbreviated BCET and WCET.

2. Related Work

For WCET estimation, flow analysis, low-level analysis and calculation phases of the program need to be considered. Flow analysis phase or the high-level analysis helps to extract the program's dynamic behavior that gives information about the addressed functions, number of loop iterations, dependencies among if-statements, etc. As the analysis is unaware of the execution path corresponding to the longest execution time, it requires such information to give a safe approximation and must include the possible program executions. There are methods to procure the information such as *manual annotations* [4], *automatic flow analysis* [8]-[12], or obtained separately [5]-[7]. The analysis is usually done on the source code; machine or intermediate code level can be used. In case of global low-level analysis different caches such as the instruction caches [7], [9], [12]-[14], data caches [12], [15], [16], and branch predictors [17], [18] are analyzed. As compared to this, the local low-level analysis deals with scalar pipelines [9], [11]-[12], [14], [17], [19]-[21] as well as superscalar CPUs [22], [23]. Authors in [13] have argued that an integration of cache and pipeline analysis is essential for processors involved with heavy interdependencies among elements performing different functions. To extract timing information, measurements and hardware have been used in [24]. The calculation phase estimates the WCET of a program by combining previous phase flow and timing information [1]-[12].

WCET-optimizing instruction set selection bears similarity to other static optimizations targeting the worst-case path like instruction cache locking or scratchpad memory allocation of program codes. In this section, we will point out the differences between these problems for the selection of the WCET-optimizing instruction. We also discuss the modern solutions available for the selection of instruction sets and explain their limitations. Caches are used to effectively reduce the average memory access latency of a CPU. It is extremely difficult to predict whether a memory access can be served by the cache hit or if it needs to be served by the main memory. WCET analysis always needs to reflect a cache miss when it cannot guarantee a cache hit. This analysis directs to an over-evaluation of the WCET bound. Cache locking is a software-controlled mechanism to load code segments into the cache and prevent them from being driven out. Several works utilize instruction cache locking techniques to reduce overestimation which results from a cache analysis by lowering the WCET bound [27], [29], [30]. In the selection process of the instruction set which has several alternatives to choose from the original software or different CIs, where implementation of the same functionality with the various degrees of parallelism and resource requirements with the extensions for evaluating multiple alternatives to choose from (e.g., the different CI implementations), the existing algorithms for cache locking would remain inappropriate for our problem. Falk et al. [27] and Liu et al. [29] model the problem similarly using Execution Flow Graphs and Execution Flow Trees, respectively. However, the execution flow is modeled on the level of function calls. Yu and Mitra [32] perform WCET-optimizing instruction set selection for extensible processors. These processors contain custom functional units that can be configured to implement the frequently used instruction pattern, which speeds up the process by exploiting instruction level parallelism and operator chaining. The WCET-optimizing instruction set is selected per task, that is, during task execution the instruction set is fixed. Therefore, the cost of configuring a selected pattern is not taken into account during the execution of this approach. In our work, we mainly target a dynamic reconfiguration based custom instructions with varying area demands (1 up to A units of the reconfigurable fabric area). In order to evaluate the profit of an instruction by reducing the WCET estimate, we need a factor that requires a demand in area as well as its reconfiguration delay. The impact of the reconfiguration delay on WCET optimization is evaluated in section 5.

The application-specific instruction set architectures (ISAs) synthesis can be accomplished in a number of ways based on the first part of ISA to be decided, IS-oriented as well as structure-oriented. Optimization of IS using the application's behavior is made with IS-oriented methods [13]-[15]. The method is based on dependency graphs. To implement the instruction set either manually or automatically, hardware design is made afterwards. Among different approaches the PEAS-I found to be similar to our method as both these methods consider the basic IS and target pipelined RISC architectures. Nevertheless, the PEAS-I involves a fixed set of instructions to select a subset, unlike our method, instruction encoding is not an issue here. Other approaches cannot be applied easily to modern pipelined RISC processors due to use of individually designed architectural styles such as the transport triggered architecture.

Due to increasing complexity, high computation performance and manufacturing costs in line with rapid development of advanced integrated circuit (IC) technology, the demand for high-performance configurable designs has surfaced. It requires the ASIPs to be incorporated with the SOC design more frequently. A most common technique in this regard is the generation of automated software tools to suit ASIPs. However, to implement the final RTL, this method is seldom applied. For better ASIP design, the consumption time needs to be reduced to satisfy the constrained time-to-market requirements. There have been other alternate architectural solutions that use the design, flexibility, as well as the time to market altogether. One of those approaches uses a low-cost GPP core with domain specific instructions. This type of processor architecture is termed as the ASIP that improves the application performances such as the throughput, latency, etc., efficiently.

Authors have proposed different ASIP design flows and ASIP case studies [16]-[26]. Their work includes processor customization, processor identification and instantiates custom instructions in a processor. This paper proposes an alternative approach, which can produce better architectures involving hardware complexity. It aims to reduce the area cost in comparison with the original soft-core processor. It analyzes the application's source code initially for identification of redundant processor instructions and removes them prior to the logical synthesis process.

Application Specific Instruction set Processor (ASIP) is a comparatively new approach to realize programmable processors, which for the targeted application domain can deliver very significant performance and power benefits, while regaining the advantage of functional flexibility through software programmability. In a sense, ASIPs bridge the design space between general-purpose processor based implementation of the application and dedicated hardware implementation of the same application as an ASIC.

ASIP Design Space has two basic classes named as (a) Instruction Set Architecture (ISA) based processor, but customized for application and programming in a sw-design type of task and (b) Programmable-HW based. It may be of FPGA programming for a HW-design type of task. It must be suitable for data flow–like computation. ASIP design is not yet disciplined, but a "form of art". Some of the gaps in ASIP methodology are: (i) incomplete application characterization;(ii) ad-hoc architectural exploration; designer's expertise; and (iii) poor software environments, especially compilers.

3. Methodology

3.1. Static Analysis techniques

To estimate WCET, the static tool examines the computer software instead of executing it on the hardware directly. Although static analysis was a prominent research method during late 1980s, end-to-end measurements offer a standard approach in the industrial setting. Both tools of static analysis perform at a high-level in order to determine the program structure, which works on source code, or executable disassembled binary. It is possible to work at a low-level provided the real hardware timing information for the executable task having the desired features. The presence of architectural features complicates the analysis of static WCET at low-level and improves a processor's average-case performance that includes the branch prediction, instruction caches and instruction pipelines. In timing models it is still difficult to achieve tight WCET bounds in case these advanced architectural features are considered. For example, to simplify WCET estimation for better predictability cache-locking techniques have been used.

3.2. System Model and Problem Formulation

Our optimization approach is applied to Control-Flow Graph (CFG) of an application in the binary format, as it is the only way to achieve secure and precise WCET estimate values. The granularity of a CI, that is, the amount of software it replaces, depends on the specific target architecture. For configuring the CIs in hardware, we assume reconfigurable fabric area to be allocable in up to A discrete units. This corresponds to the common area model of dividing the fabric area into equally sized partitions like in the 1D or 2D partitioned area models in Steiger et al. [31]. Let CI be the set of all CIs. We assume a specific configuration j of a CI $k \in CI$ in hardware to have a constant delay $t_{k,j}$ to require area on the reconfigurable fabric $a_{k,j} \in$ [1], and to take a constant reconfiguration delay $r_{k,j}$ for configuring it on the fabric. For a constant reconfiguration delay, a constant bandwidth for transferring configuration data to the reconfigurable fabric's configuration memory needs to be guaranteed. We assume the CPU to be delayed during reconfiguration in this work, and therefore the system bus could be utilized for reconfiguration at a guaranteed bandwidth. Along with hardware configurations, a CI can be implemented using its original software code j = 0. Since it has been implemented with a software, it does not have a constant delay $t_{k,0,}$ because of specific cache and pipeline analysis (i.e., $a_{k,0} = r_{k,0} = 0$).

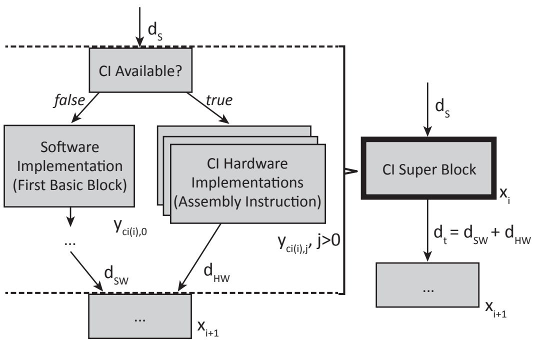

Fig . 2. CI super block as part of a CFG
Reference: [22-23]

In order to provide flexibility to execute the original software for generated CIs, we introduce CI super blocks. As shown in Fig. 2, CI superblocks begin with a conditional branch before every CI (the actual instruction in the binary); which jumps to the functionally equivalent software code when the CI is not implemented in hardware. If a configuration for the CI is available on the reconfigurable fabric, then it is executed instead of jumping to the software. The CI super block ends by joining paths of hardware CI and software. Multiple CI superblocks in the binary can execute the same CI k. Let B be the set of all blocks, that is, basic blocks (not contained in super blocks) as well as super blocks. The function ci(i) determines which CI k is executed by a super block i ∈B, that is,

ci: B → CI∪ {0}, i → k, with ci(i) = 0 ∈CI if i is a basic block (not a super block)

The context-dependent delay for executing the implementation j of CI super block i is denoted as $e_{i,j}$ for both hardware and software implementations. While CI execution on the reconfigurable fabric itself is context independent ($t_{ci}(i)$, j is constant, for j > 0), invoking the CI from the CPU pipeline can add additional cycles, for example, because of pipeline hazards present in CI. Therefore, $e_{i,j} \geq t_{ci(i),j}$ for j > 0. Equation (1) is used to concisely formulate equations on which our WCET estimation is based. Effectively, we obtain a CFG that can be parameterized by a chosen selection using CI super blocks. In the following, we will introduce the WCET bound estimation technique we utilize and show how we can extend it to our problem formulation to evaluate and direct our optimization.

Selecting an instruction set to optimize the WCET bound essentially means we aim to minimize the WCET over all possible selections, that is, we aim to minimize the maximum execution time. We extend the ILP formulation of IPET for capturing the implementation alternatives of a CI*k* ∈CI. We introduce new variables $y_{k,j} \in \{0, 1\}$ for every implementation *j* with $y_{k,j} = 1$ if CI*k* is implemented using alternative *j* and $y_{k,j} = 0$ otherwise.

$$\textstyle\sum_{0<j<n} y_{k,j} = 1 \tag{1}$$

The total cycle contribution of CI*k*'s super block *i* to the WCET bound is given as follows:

$$\sum\nolimits^{\square}_{\substack{1\leq i\leq|B| \\ 0<j<me(i)}} \square \tag{2}$$

The WCET for a given selection *y* without accounting for reconfiguration delay can be determined as follows:

$$\mathrm{WCET}\ (y) = \begin{matrix} B \vee e_{i,j} y_{c(i),j}, x_i \\ ci x\mathrm{i} + \sum\nolimits^{\square}_{\substack{1\leq i\leq|B| \\ 0<j<me(i)}} \square \\ \sum\nolimits^{|B|}_{\substack{i=1 \\ Ci(i)\notin Cj}} \square \\ maxx \end{matrix} \tag{3}$$

Every CI super block utilized in a kernel is configured exactly once before entering the kernel (with zero reconfiguration delay for software implementation). Therefore, we obtain the WCET including reconfiguration delay as:

$$\text{WCET}(y) = \text{WCET}'(y) + \sum_{\substack{0\le i\le m \\ 0<j<n}} y_{k,j}\, r_{k,j} \qquad (4)$$

Putting it all together, the WCET-optimizing instruction set selection problem becomes a combinatorial problem with the following objective function:

$$\begin{gathered} B \vee e_{i,j} y_{c(i),j}, x_i \\ cixi + \sum^{\square}_{\substack{1\le i\le |B| \\ 0<j<me(i)}} \square \\ maxx\left(\sum^{|B|}_{\substack{i=1 \\ Ci(i)\notin Cj}} \square + \sum_{\substack{0\le i\le m \\ 0<j<n}} y_{k,j}\, r_{k,j}\right) \\ y[0,1] \\ min \end{gathered} \qquad (5)$$

However, this would result in up to $2^{|CI|*M}$ constraints of high complexity, which becomes practically infeasible even for small values. Also, note that we do not need to evaluate the ILPS for the IPET instance of the whole application, but only per kernel. Therefore, ILPS are considerably less complex (fewer variables and constraints) than the ILP for determining the WCET of the whole application. In the following section, we will show how the search space can be pruned and feasible y is generated efficiently.

3.3. Discrete Particle Swarm Optimization (DPSO)

The PSO remains a derivative-free global optimized algorithm having no Hessians or gradients to compute. The DPSO applies qualitative or discrete distinction between the designated variables and is the modified PSO. Kennedy and Eberhart [24] developed the first DPSO with binary valued particles. Since then several versions of DPSO have been developed. DPSO will facilitate solving the combinatorial optimization problems due to its ease of implementation, simple structure and its robustness. In the DPSO algorithm of OP each particle constitutes a tour encompassing the list of nodes visited such that *Tmax*, the distance constraint was obeyed. The starting and ending nodes are distinct and specified. To ensure a good starting solution in the population, the first particle was built using the *s/d* (score/distance) ratio. Beginning from the first node, the feasible node having the highest *s/d* value is selected as the next city that needs to be visited. Construction of the first particle is based on these *s/d* values as the feasible tour. The initial solutions for the remaining particles were constructed randomly [27-28].

The new position (new tour) for each particle was calculated as follows: Generate a random number Λ corresponding to each node in the current particle. If $\Lambda < w$, accept the node as a part of a temporary array, indicated as Ω. Include similarly nodes in the temporary array using the *pi best* particle in case $\Lambda < c1$ else use the *g best* if $\Lambda < c2$. This process removes the duplicate nodes from Ω. Based on the feasibility of the temporary array, the new position is either accepted or randomly deleted in the particle until a feasible solution is obtained. For better efficacy and to avoid the algorithm to fall in local optima, researchers have used a local search tool such as the Reduced Variable Neighborhood Search (RVNS). On switching neighborhoods, the solution space in RVNS is searched. These two neighborhoods of the algorithm are *insert* and *exchange*. While a new node is put in the insert neighborhood a in the existing tour, whereas it is exchanged in the exchange neighborhood for an existing node corresponding to the tour. On completion of RVNS, optimization of the tour is performed using a 2-opt operation. A node insert operation is carried out in case the tour length is reduced by the 2-opt method. It attempts to increase the total score or fitness value of the tour. The DPSO algorithm flowchart has been provided in Fig. 2.

In the DPSO, the particle values are restricted either to 0s or 1s. They can be applied for solving problems with integer variables. Initially, in a DPSO algorithm, a pool of solutions is randomly generated. In the solution pool, the position vector describes each solution

$$X_n = (x_{n1}, x_{n2}, x_{n3}, \dots \dots x_{nd}) \qquad (6)$$

Where n is the swarm size or the number of solutions (n = 1, 2, . . ., N) whereas, d is the binary equivalent of variables in the solution space. The length of the binary solution X_n is selected based on the number of variables as well as the binary bits corresponding to each variable. Each solution is associated with a velocity vector represented by

$$V_n = (v_{n1}, v_{n2}, v_{n3}, \ldots \ldots v_{nd}) \quad (7)$$

During each iteration t (t = 1, 2, . . ., T), the dimensions of each velocity vector V_n have been updated by the equation

$$v_{nd}^{t+1} = \omega v_{nd}^{t} + \theta_1(b_{nd}^{t} - x_{nd}^{t}) + \theta_2(g_d^{t} - x_{nd}^{t}) \quad (8)$$

here, b_{nd} and g_d are the local and the global best positions respectively as achieved by the constituent particles. The notation ω signifies the inertia weight and is a constant similar to the learning factors θ_1 and θ_2. The learning factors indicate the importance given to the local best or the global best in the swarm and are based on experience. To find the new position of x_{nd} (0 or 1) equation (8) has been used, where the random variable r is generated uniformly between the values 0 and 1.

$$x_{nd}^{t+1} = \begin{cases} 1, if r < S(v_{nd}^{t+1}) \\ 0, otherwise \end{cases} \quad (9)$$

In this way, a swarm of size N is generated by iterating T time using the above procedure each time to achieve the optimal or near optimal solution.

3.4. Optimal Solution using PSO

In theory, we can have up to $2^{|CI|*M}$ possible selections y that we need to evaluate. In practice, however, the search space is considerably smaller for the following reasons: The number of possible hardware configurations m_k per CI_k varies a lot, for example, in our evaluation we had a minimum of 1 to a maximum of 78 = M implementations for CIs (including software implementation) within one kernel. For the CIwith 78 different implementations, many implementations had different degrees of parallelism and latencies but required the same amount of area and reconfiguration delay when synthesized to the reconfigurable fabric. When considering only the minimum-latency implementation per required fabric area, our algorithm was able to prune the number of implementations to relevant ones. Therefore, in practice the relevant number of implementations per CI_k is much smaller than m_{k+1}.

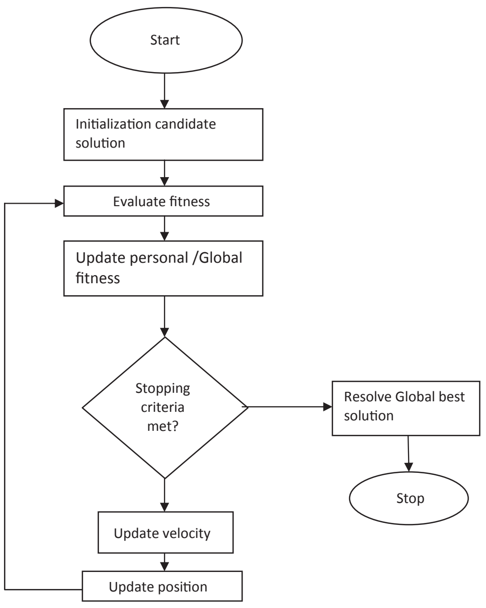
Start
Initialization candidate solution
Evaluate fitness
Update personal /Global fitness
Stopping criteria met?
Resolve Global best solution
Stop
Update velocity
Update position

Fig. 3. Flow chart for DPSO based optimization procedure Reference: [24]

The implementation y^{+k} *for a* CI_k is the implementation that can be chosen with minimum increase for an area over *yk* resulting in a positive profit. There might be several implementations according to this definition with the same required area. In this case, y^{+k} is the implementation with minimum latency $t_{k,j}$ (and min *j*). If no such implementation y^{+k} exists (i.e., no implementation with positive profit was found), then the CI is not considered for selecting a different implementation. Among the CIs for which y^{+k} exists, the algorithm PSOchooses the one with the maximum profit and upgrades *y* to select y^{+k} for the chosen CI_k. This process is repeated such that every iteration CI_k with maximum profit (y^{+k}, $y_{k,x}$) is upgraded. The algorithm terminates when no y^{+k} for any *k* exists anymore or insufficient area is left to be allocated for selecting y^{+k}. In every iteration, we either select a CI for increasing its allocated area by a minimum of one or the algorithm terminates for every iteration, but the last we performed was a WCET estimate. Therefore, we perform a maximum of *A* WCET estimates.

4. Results and Discussion

We evaluated this work on a reconfigurable processor presented in Henkel et al. [28] that we extended for execution with hard real-time guarantees. A reconfiguration Unit with private memory to store configurations provides predictable reconfiguration of CIS. Initiating a specific configuration is done by a single store of the CPU using the memory-mapped interface of the reconfiguration. We experimented, tested and evaluated our analysis with an H.264 encoder application that utilizes nine CIS covering the majority computation intensive kernels. Multimedia applications are frequently subject to hard real time constraints in the field of computer vision. The H.264 encoder consists of complicated control flow with more number of decisions and nested loops. Which path is executed within a kernel iteration relies upon the nature of MB, either I-MB or P-MB, determined by the motion estimation kernel, that is, it is input dependent the I-MB and P-MB paths also contain separate CIS leading to instability of the worst-case path, that is, adding more partitions to the current worst-case path can result in the other path becoming the worst case. The pipeline stalls for the reconfiguration delay and continues with entering the kernel once the reconfiguration finishes. The parameters evaluated were different numbers of s partitions A (300 slices each on a Xilinx Virtex 7), reconfiguration bandwidths as well as relations of CPU frequency and fabric frequency $f_{CPU/}f_{fabric}$. f_{fabric} remains constant at 100MHz, and we select multiple values of it for f_{CPU} that resemble realistic units. Note that while the WCET in seconds (WCET cycles/ f_{CPU}) is anticipated to get lower (better) with higher f_{CPU}, the WCET cycles are increasing (at a constant f_{fabric}), because hardware CIS perform less computations on the reconfigurable fabric within one CPU cycle [29]-[33].

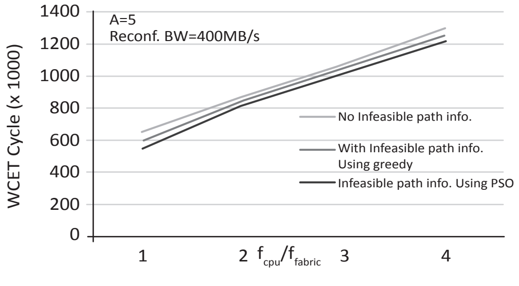

Fig . 4. Optimal results for the Encode Macro Block kernel of the H.264 encoder and different values of f_{CPU}/ f_{fabric}, A as well as reconfiguration bandwidth
Reference: [12]

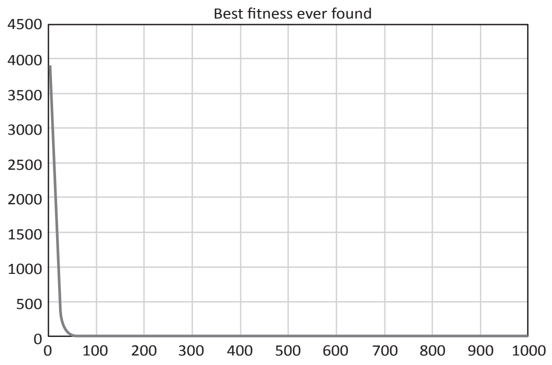

Fig . 5. Convergence graph of fitness function using DPSO
Reference: [24]

The most relevant findings should be mentioned. It is recommended, if it is the case, to use graphs and tables to synthesize the information. If formulas were used, it is convenient to present them and explain their importance in the study. The results presented must be focused on the question and focus of the investigation. It is important to present them in an orderly, specific way and without personal comments or appreciations. Should be noted, if appropriate, observations, experiments and data obtained throughout the investigation.

5. Conclusion and Future Work

This paper presents a novel method to estimate the WCET of a program. The method is similar to the hybridization of tree and path-based methods, which are less precise and global IPET method, which is more precise. It determines how to obtain the smallest feasible parts of a program, which need to be handled as an entity for better precision. The method of computation to be applied for each such part has not been fixed since it is based on the chosen flow of information and program structure characteristics. As these parts are generally small in comparison with the overall program, the method remains fast with no loss of precision on account of introducing arbitrary boundaries during computation in tree and path-based approaches.

References

[1] S. S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki, "An Accurate Worst-Case Timing Analysis for RISC

Processors," *IEEE Transactions on Software Engineering,* vol. 21, no. 7, pp. 593-604, Jul. 1995. doi: https://doi.org/10.1109/32.392980

[2] S. K. Kim, S. L. Min, and R. Ha, "Efficient Worst Case Timing Analysis of Data Caching," in *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96). IEEE,* 1996. doi: https://doi.org/10.1109/rttas.1996.509540

[3] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," in *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97),* Jun 1997, pp. 192-202, doi: https://doi.org/10.1109/RTTAS.1997.601358

[4] A. Colin and I. Puaut, "Worst Case Execution Time Analysis for a Processor with Branch Prediction," *Journal of Real-Time Systems,* vol. 18, no. 2/3, pp. 249-274, May 2000, doi: https://doi.org/10.1023/a:1008149332687

[5] T. Mitra and A. Roychoudhury, "Effects of Branch Prediction on Worst Case Execution Time of Programs," National University of Singapore (NUS), Tech. Rep. 11-01, Nov 2001.

[6] J. Engblom, "Processor Pipelines and Static Worst-Case Execution Time Analysis," Ph.D. dissertation, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, Apr. 2002. Available: https://www.diva-portal.org/smash/get/diva2:161408/FULLTEXT01.pdf

[7] J. Engblom and A. Ermedahl, "Pipeline Timing Analysis Using a Trace-Driven Simulator," in *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99). IEEE Computer Society Press,* Dec1999. doi: https://doi.org/10.1109/RTCSA.1999.811197

[8] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and Precise WCET Determination for a Real-Life Processor," in *Proc. 1st International Workshop on Embedded Systems, (EMOSOFT2000), LNCS2211*, Oct. 2001.

[9] S. S. Lim, J. H. Han, J. Kim, and S. L. Min, "A Worst Case Timing Analysis Technique for Multiple-Issue Machines," in *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98),* Dec 1998. doi: https://doi.org/10.1109/REAL.1998.739765

[10] J. Schneider and C. Ferdinand, "Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation," in *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99*). May 1999. doi: https://doi.org/10.1145/315253.314432

[11] S. Petters and G. Farber, "Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible," in *Proc. 6th International Conferenceon Real-Time Computing Systems and Applications (RTCSA'99)*, Dec. 1999. doi: https://doi.org/10.1109/RTCSA.1999.811296

[12] M. Venkanna, R. Rao, and P.Chandra Sekhar, "Application of ASIP in Embedded Design with Optimized Clock Management," *ICITKM Conference*, Newdelhi, pp. 161-165, 2017. doi: https://doi.org/10.15439/2017km41

[13] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai, "PEAS-I: A hardware/software co-design system for ASIPS," in *Proc. EURO-DAC*, 1993. doi: https://doi.org/10.1109/EURDAC.1993.410608

[14] J. Van Praetet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction selection for ASIPS," in *Proc. HLS Symposium 1994, Instruction set definition and instruction selection for ASIPs*, 1994, doi: https://doi.org/10.1109/ISHLS.1994.302348.

[15] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration through Automated Instruction Set Customization." In *Proceedings of the 36th annual IEEE/ACM International Symposium on Micro architecture (MICRO36)*, 2003.

[16] R. R. Hoare et al., "Rapid VLIW Processor Customization for Signal Processing Applications Using Combinational Hardware Functions," *EURASIP Journal on Applied Signal Processing,* vol. 2006, no. 46472, 2010. doi: https://doi.org/10.1155/ASP/2006/46472

[17] F. Tlili and A. Ghorbel, "ASIP Solution for Implementation of H.264 Multi Resolution Motion Estimation," *International Journal of Communications, Network and System Sciences,* vol. 3 no. 5, May 2010. doi: https://doi.org/10.4236/ijcns.2010.35060

[18] P. Guironnet de Massas, P. Amblard, and F. Pétrot, "On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration," *Journal VLSI Design,* vol. 2007, no. 28686, 2007. doi: https://doi.org/10.1155/2007/28686

[19] S. Tillich, "Instruction Set Extensions for Secret-Key Cryptography," *Ph .D. Forum at the 9th Conference on Design, Automation and Test in Europe (2006)*, Munich, Germany, March 6, 2006. doi: https://doi.org/10.1109/CCST.2014.6986988

[20] F. Naessens, A. Bourdoux, and A. Dejonghe, "A flexible ASIP decoder for combined binary and non-binary LDPC codes," *17th IEEE Symposium on Communications and Vehicular Technology (SCVT),* 24-25 Nov. 2010. doi: https://doi.org/10.1109/SCVT.2010.5720462

[21] G. Kappen, L. Kurz, O. Priebe, and T. G. Noll, “Design Space Exploration for an ASIP/Co-Processor Architecture used in GNSS Receivers,” *Journal of Signal Processing Systems,* vol. 58, no. 1, pp. 41-51, 2010. doi: https://doi.org/10.1007/s11265-008-0261-z

[22] J. Kennedy and R. C. Eberhart, “A discrete binary version of the particle swarm algorithm,” in *IEEE International Conference on Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation*, 1997, vol. 5, pp. 4104-4108. doi: https://doi.org/10.1109/ICSMC.1997.637339

[23] Q. K. Pan, M. F. Tasgetiren, and Y. C. Liang, “A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem,” *Computers & Operations Research*, vol. 35, no. 9, pp. 2807-2839, 2008. doi: https://doi.org/10.1016/j.cor.2006.12.030

[24] H. Falk and J. C. Kleinsorge, “Optimal static WCET-aware scratchpad allocation of program code,” *In Proc. of Design Automat. Conf. ACM,* pp. 732-737, 2009. doi: https://doi.org/10.1145/1629911.1630101

[25] H. Falk, S. Plazar, and H. Theiling, “Compile-time decided instruction cache locking using worst-case execution paths,” in *Proc. of Int. Conf. on Hardware/Software Codesign and Syst. Synthesis. ACM*, pp. 143-148, 2007. doi: https://doi.org/10.1145/1289816.1289853

[26] J. Henkel, L. Bauer, M. Hubner, and Ar. Grudnitsky, “i-Core: A run-time adaptive processor for embedded multi-core systems”, in *Proc. Int. Conf. on Engineering of Reconfig. Syst. and Algorithms*, 2011. Available: http://worldcomp-proceedings.com/proc/p2011/ERS6061.pdf

[27] T. Liu, M. Li, and C. J. Xue, “Minimizing WCET for real-time embedded systems via static instruction cache locking”, in *Real-Time and Embed. Technol. Applications Symp. IEEE*, pp. 35-44, 2009. doi: https://doi.org/10.1109/RTAS.2009.11

[28] S. Plazar, J. C. Kleinsorge, P. Marwedel and H. Falk, “WCET-aware static locking of instruction caches”, in *Proc. 10th International Symposium on Code Generation and Optimization, ACM,* pp. 44-52, 2012. doi: https://doi.org/10.1145/2259016.2259023.

[29] C. Steiger, H. Walder, M. Platzner, and L. Thiele, “Online scheduling and placement of real-time tasks to partially reconfigurable devices,” in *Proc. of Real-Time Syst. Symp. IEEE*, pp. 224-225, 2003. doi: https://doi.org/10.1109/REAL.2003.1253269

[30] P. Yu and T. Mitra, “Scalable custom instructions identification for instruction-set extensible processors,” in *Proc. of Int. Conf. on Compilers, Architecture and Synthesis for Embed. Syst. ACM,* pp. 69-78, 2004. doi: https://doi.org/10.1145/1023833.1023844