Verteilte HW/SW-Partitionierung für fehlertolerante rekonfigurierbare Netzwerke *

Thilo Streichert, Christian Haubelt, Jürgen Teich
Universität Erlangen-Nürnberg
Lehrstuhl für Hardware-Software-Co-Design
Am Weichselgarten 3, 91058 Erlangen, Germany
{streichert,haubelt,teich}@cs.fau.de

Abstract

Eingebettete Systeme bestehen heutzutage immer häufiger aus vernetzten Ressourcen, deren Funktionalität aus Gründen der Verlässlichkeit und Fehlertoleranz verteilt im Netzwerk implementiert ist. Der höhere Entwurfsaufwand für solche verteilten Architekturen rechnet sich, sofern diese Systeme durch redundante Implementierungen der Prozesse, den Grad an Fehlertoleranz erhöhen. Dies bedeutet insbesondere, dass solche Systeme auf Ressourcedefekte durch erneutes Binden der Prozesse auf funktionsfähige Einheiten reagieren und die Funktionalität des Systems garantieren kann. Hardware (HW)/Software (SW) rekonfigurierbare Systeme *bieten hier die Möglichkeit, bestehende Anforderungen an Geschwindigkeit, Parallelität und Leistungsverbrauch mit Flexibilität zu vereinen. In diesem Beitrag behandeln wir Systeme, die adaptiv auf Laufzeitanforderungen reagieren und die Rechenlast im Netzwerk auf die einzelnen Ressourcen verteilen. Bislang gibt es keine Methoden, die dieses Problem bestehend aus einer HW/SW-Partitionierung und einer Lastbalanzierung zur Laufzeit behandeln. Im folgenden Beitrag werden Gründe für dieses Vorgehen geliefert und experimentelle Analysen für die vorgeschlagene Vorgehensweise präsentiert. Des Weiteren gibt es eine erste Implementierung des Konzepts bestehend aus einem Netzwerk mit vier FPGA-basierten Boards.*

1. Einleitung

Verteilte und adaptive eingebettete HW-Plattformen können in naher Zukunft eine interessante Alternative für bestehende Systeme darstellen, wie z.B. Anwendungen des Automotive-Bereichs, Body-Area-Netzwerke oder "Ambient Intelligence". Damit solche Systeme mit zeitvarianten Anforderungen umgehen können, müssen sie adaptiv sein und auf unvorhersehbare Ereignisse reagieren können. Aus diesem Grund soll Funktionalität, die in HW oder SW implementiert ist, dynamisch den Ressourcen im Netzwerk zugeordnet werden. Derzeitige Offline-Partitionierungsalgorithmen können lediglich optimale Lösungen bezüglich vorgegebener Ziele, wie Kosten, Fläche, Leistung, usw. für statische Anwendungen und Architekturen finden.

*mit Unterstützung der Deutschen Forschungsgemeinschaft (DFG) TE 163/10-1, SPP 1148 (Rekonfigurierbare Rechensysteme)

In diesem Beitrag stellen wir uns der Herausforderung, Online-Methoden für verteilte eingebettete Systeme zu entwickeln, so dass ein solches System auf Defekte optimal reagieren kann. Hierbei betrachten wir weniger den Aspekt der redundanten Ausführung von Prozessen als vielmehr die Fragestellung, ob ein solches System eine optimale Verteilung seiner Tasks nach einem Defekt finden kann. Eine Arbeit [8] in diesem Gebiet beschreibt Lastbalanzierung auf einer Plattform mit einem Prozessor und rekonfigurierbarer HW, jedoch gibt es keine Erweiterung für Netzwerke aus solchen Plattformen. Andere Arbeiten behandeln entweder nur Offline-HW/SW-Partitionierung [7] oder nur Lastbalanzierungsalgorithmen [5]. Sie betrachten aber nicht die Fragestellung, wie zur Laufzeit entschieden werden kann auf welchem Knoten und in welcher Implementierungsart - HW oder SW - Funktionalität ausgeführt wird.
Die Gliederung dieses Beitrags ist wie folgt: In Abschnitt 2 stellen wir das Online-Partitionierungsproblem vor und erläutern unser Modell für die formale Beschreibung von Systemen und deren Funktionalität. Im ersten Teil von Abschnitt 3, erklären wir die Arbeitsweise von Diffusionsalgorithmen und erläutern, wie wir diesen Algorithmus diskretisiert haben. Bezüglich der Güte des diskretisierten Algorithmus, zeigen wir Grenzen auf, die theoretisch beweisbar sind [9]. Anschließend präsentieren wir die Bipartitionierung, die eine optimale Partitionierung bezüglich verschiedener Zielgrößen finden soll. Im Abschnitt 4 bewerten wir unsere lokale verteilte Partitionierungsstrategie. Der Abschnitt 5 stellt eine erste Implementierung der Algorithmen vor, die mit FPGA-basierten Boards arbeitet.

2. Modelle und Szenarien

Im Folgenden betrachten wir eingebettete Systeme, die aus HW/SW rekonfigurierbaren Knoten bestehen und über Punkt-zu-Punkt-Verbindungen kommunizieren. Dabei bedeutet HW/SW-Rekonfigurierbarkeit, dass jeder Knoten im Netzwerk einen Prozess entweder auf SW-Ressourcen sequentiell mit anderen Prozessen oder auf HW-Ressourcen parallel mit anderen Prozessen ausführen kann. Es gibt im Wesentlichen zwei Szenarien, bei denen HW/SW-(Re)Partitionierung notwendig wird: Zum einen kann in einem Netzwerk eine Kommunikationsverbindung oder ein Knoten ausfallen. Zum anderen können neue Prozesse hinzukommen und alte wegfallen, wodurch sich Ungleichge-

wichte bei der Lastverteilung zwischen den Knoten einstellen. In beiden Fällen bietet die HW/SW-Partitionierung eine Möglichkeit zur Vermeidung von Überlast auf Knoten und zur Steigerung der Fehlertoleranz.

Zunächst wird allerdings ein geeignetes Modell benötigt, anhand dessen die Vorgehensweise mathematisch beschrieben werden kann. Das Modell besteht aus einem so genannten Architekturgraph $g_a = (N, C)$, bei dem $N = \{n_1, n_2, ..., n_{|N|}\}$ die Menge der Netzwerkknoten und $C \subseteq N \times N$ der Kommunikationsverbindungen in dem Netzwerk ist. Die Menge der Prozesse $P = \{p_1, p_2, ..., p_{|P|}\}$ beinhaltet alle ausführbaren Prozesse im Netzwerk, wobei $P(t) \subseteq P$ alle aktiven Prozesse zum Zeitpunkt t beinhaltet.

Definition 1 *Das temporale HW/SW-Partitionierungsproblem ist eine Zuordnung von jedem Prozess $p \in P(t)$ zu einer Ressource $n \in N(t)$, sowie eine Zuordnung zu der HW- bzw. SW-Ressource. Im Folgenden bezeichnen wir die aktive Menge der Ressourcen und Verbindungen zum Zeitpunkt t als* temporale Allokation $\alpha(t)$. *Ebenfalls bezeichnen wir die Zuordnung von Prozessen zu Ressourcen als* temporale Bindung $\beta(t) \subseteq P(t) \times N(t) \times \{H, S\}$, *wobei H und S die Prozesse kennzeichnen, die in HW bzw. SW implementiert sind.*

Zusätzlich haben Prozesse gewisse Eigenschaften, die in eingebetteten System nicht zu vernachlässigen sind. Sie können z.B. periodisch oder sporadisch sein oder auch unterschiedliche Lasten und Deadlines haben [1]. Auf jeden Fall ist es für Online-HW/SW-Partitionierung unumgänglich eine präzise Definition der HW/SW-Last zu haben:

Definition 2 *Die SW-Last $w_i^S(p)$ auf einem Knoten n_i des Prozesses p ist das Verhältnis der Ausführungszeit zur Periode. Diese Definition kann für periodische und präemptive Prozesse verwendet werden. Buttazzo und Stankovic [2] schlagen eine Definition der SW-Last vor, bei der die Last dynamisch zur Laufzeit bestimmt wird, was im Folgenden nicht berücksichtigt wird.*

Die HW-Last $w_i^H(p)$ wird als Verhältnis der benötigten Fläche eines Prozesses und der maximal verfügbaren Fläche angesehen, wobei Fläche auch durch Logikelemente bei FPGAs ersetzt werden kann.

Beispiel 1 *Abbildung 1 zeigt ein eingebettetes rekonfigurierbares Netzwerk zu zwei Zeitpunkten t_1 und t_2. Zwischen diesen beiden Zeitpunkten fällt der Netzwerkknoten n_2 aus. Anschließend migriert der Prozess p_2 zum Ressourceknoten n_4. Schwarze Knoten repräsentieren HW-Implementierungen und weiße Knoten SW-Implementierungen. Zum Zeitpunkt t_2 kommt zusätzlich ein neuer Prozess p_6 auf dem Knoten n_3 hinzu, so dass die Allokation $\alpha(t)$ und die Bindung $\beta(t)$ von Prozessen auf Knoten sich ändert. Auf Grund der neuen Bindung von Prozessen im Netzwerk kann nicht sichergestellt werden, dass alle Prozesse innerhalb ihrer Periode ausgeführt werden. Folglich besteht die Möglichkeit, dass Deadlines verletzt werden.*

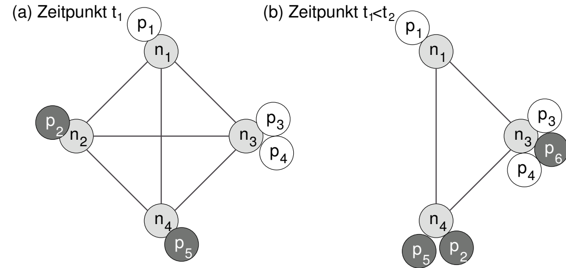

Abbildung 1. Netzwerkzustand zum Zeitpunkt t_1 und $t_2 > t_1$.

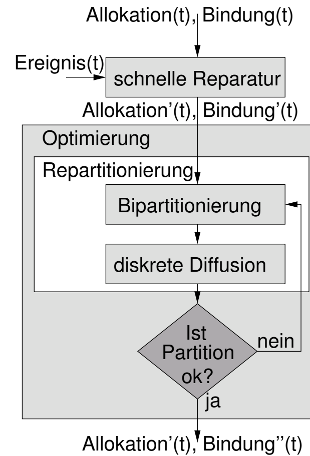

Abbildung 2. Phasen der Online-HW/SW-Partitionierung

Das Ziel der im Folgenden vorgestellten Architektur ist die Erhöhung der Fehlertoleranz bzw. die Bestimmung einer optimalen Bindung für den Fall, dass ein Knoten ausfällt oder ein neuer Prozess gestartet wird. Um diese optimale Bindung zu erhalten, schlagen wir eine zweistufige Strategie vor, die aus einer schnellen Reparatur und einer Optimierungs- bzw. (Re)Partitionierungsphase besteht, siehe Abbildung 2. In dem schnellen Reparaturschritt soll zunächst die Funktionalität des Netzwerks wiederhergestellt werden, so dass innerhalb vorgegebener Deadlines der Betrieb der Netzwerkfunktionalität ungestört fortgesetzt werden kann. In dem folgenden Schritt soll der Netzwerkzustand so angepasst werden, dass eine erneute schnelle Reparatur durchgeführt werden kann. Diese Optimierungsphase wird auch ausgeführt, wenn neue Prozesse hinzukommen. Bei beiden Strategien der ersten und zweiten Phase ist es aus Gründen der Fehlertoleranz wichtig, dass die Algorithmen verteilt im Netzwerk laufen. Im Folgenden wird nur die Optimierungsphase näher betrachtet, d.h. wir verwenden bestehende Verfahren aus der Literatur, um Defekte im System schnell zu reparieren [6].

3 Online-HW/SW-Partitionierung

Die HW/SW-Partitionierung versucht eine gegebene Bindung $\beta(t)$ von Prozessen auf Ressourcen unter der Berücksichtigung gewisser Zielgrößen zu finden. Für die

hier vorgestellten Szenarien sind insbesondere die folgenden drei Zielgrößen interessant:
Lastbalance im Netzwerk: Mit dieser Zielgröße, die es zu minimieren gilt, wird die Last zwischen den Knoten balanciert. Hierbei werden SW- und HW-Lasten getrennt behandelt: $max(max(w_i^S) - min(w_i^S), max(w_i^H) - min(w_i^H))$ mit $w_i^{\{H,S\}} = \sum_{\forall p \in P(t)} w_i^{\{H,S\}}(p)$.
HW/SW-Lastbalance: Durch die zweite Zielgröße soll eine Bipartition gefunden werden, so dass die Last von HW und SW-Prozessen auf einem Knoten gleichmäßig verteilt ist. Es soll also folgende Funktion minimiert werden: $\left| \sum_{i=1}^{|N|} w_i^{\mathrm{S}} - \sum_{i=1}^{|N|} w_i^{\mathrm{H}} \right|$. Mit Hilfe der ersten beiden Zielgrößen soll die Lastreserve auf jedem Knoten für jede Implementierungsart maximiert werden, so dass bei zukünftigen Knotenausfällen kurze Reparaturzeiten erreicht werden und Prozesse z.B. immer von einem Nachbarknoten aufgenommen werden können.
Minimierung der Gesamtlast: Falls diese Zielgröße nicht berücksichtigt wird, kann ein Algorithmus eine Verteilung finden, die optimal bezüglich der ersten beiden Zielgrößen ist, aber eine hohe Rechenlast im Netzwerk erzeugt. Es gilt also $\sum_{i=1}^{|N|} w_i^S + w_i^H$ zu minimieren.
Um diese drei Zielgrößen zu erfüllen, bietet sich eine HS/SW-Partitionierungsstrategie an, die aus einer Bipartitionierungsphase besteht und einem Lastbalanzierungsalgorithmus, siehe Abbildung 2. Während der Lastbalanzierungsschritt die erste Zielgröße verfolgt, versucht die lokale Bipartitionierung eine gute Aufteilung der Prozesse auf HW- bzw. SW-Ressourcen bezüglich der zweiten und dritten Zielgröße zu finden.

3.1 Lastbalanzierung

Für die Lastbalanzierung findet ein Diffusionsalgorithmus Anwendung, der zwischen adjazenten Netzwerkknoten Lasten verschiebt. Charakteristisch für Diffusionsalgorithmen, die von Cybenko [3] vorgestellt wurden, ist die Eigenschaft, dass iterativ und alternierend ein reellwertiger Anteil der Last eines Knotens zwischen Knoten im Netzwerk über eine Kante $c \in C(t)$ verschoben werden kann. Diese Eigenschaft ist allerdings nicht unproblematisch, da in aller Regel nur ganze Prozesse verschoben werden können. Außerdem können durch das alternierende Verhalten bei einigen Weiterentwicklungen des Diffusionsalgorithmus negative Lasten auf einem Knoten entstehen, was in der Realität nicht modellierbar ist [4].
Dennoch zeichnen sich diese Algorithmen durch ihre Konvergenzeigenschaften aus, weshalb im Folgenden ein diskreter Diffusionsalgorithmus vorgeschlagen wird, mit dessen Hilfe die beiden genannten Probleme behoben werden.

Definition 3 *Ein lokaler iterativer Lastbalanzierungsalgorithmus führt auf jedem Knoten von $n \in N$ Iterationen durch, in denen die eigene Last mit der Last der adjazenten Knoten abgeglichen wird. Hierfür wird auf jedem Knoten n_i folgende Berechnung durchgeführt:*

$$\begin{aligned} y_c^{k-1} &= \alpha(w_i^{k-1} - w_j^{k-1}) \quad \forall c = \{n_i, n_j\} \in C \\ x_c^k &= x_c^{k-1} + y_c^{k-1} \quad \forall c = \{n_i, n_j\} \in C \\ w_i^k &= w_i^{k-1} - \sum_{c=\{n_i,n_j\} \in C} y_c^{k-1} \end{aligned} \tag{1}$$

In der obigen Definition bezeichnet y_c^k die Last, die über die Kante c in Iteration k mit α als Vorfaktor übertragen wird. x_c^k ist die Gesamtlast, die bis zur letzten Iteration k über c übertragen wird. w_i^k bezeichnet die Last nach k Iterationen. In [3] wurde gezeigt, dass der Algorithmus für reellwertige Lasteinheiten y_c^k zum Lastdurchschnitt $\bar{w}$ konvergiert, wobei die Anzahl der Iterationen groß sein kann.

Eine leichte Modifikation des Diffusionsalgorithmus arbeitet mit wechselnden Werten von α in jeder Iteration k, wodurch man einen optimal balancierten Zustand in genau $m-1$ Iterationen erreicht [4]. Hierfür wählt man $\alpha = \frac{1}{\lambda_k}$ in jeder Iteration von Eq. (1). Der Parameter λ_k ergibt sich aus den nicht-trivialen Eigenwerten der Laplace-Matrix $L = D - B$, wobei die Matrix D den Grad eines jeden Knotens als Diagonalelement enthält und B die Adjazenzmatrix des Netzwerks ist.
Die als uniforme Diffusion bekannte Lastbalanzierung arbeitet mit einem konstanten Wert von α: $\alpha = \frac{1}{deg(n_i)+1}$. α ist in diesem Fall das Reziproke des um Eins inkrementierten Grads eines Knotens.
Da im ersten Fall zur Laufzeit Eigenwerte bestimmt werden müssen, ist die Skalierbarkeit eines solchen Algorithmus problematisch. Aus diesem Grund kommt die uniforme Diffusion häufiger zum Einsatz.
Damit die betrachteten Diffusionsalgorithmen in realen Anwendungen eingesetzt werden können, schlagen wir eine Erweiterung vor, welche die folgenden zwei Probleme löst:

- Die Diffusionsalgorithmen arbeiten kontinuierlich, so dass reellwertige Prozessteile zwischen den Knoten migrieren. Gesucht ist jedoch ein diskreter Algorithmus, der ganze Prozesse verschiebt.
- Diffusionsalgorithmen sind alternierend, dass heißt, dass negative Lasten auf einem Knoten vorkommen können.[1]

Für die Diskretisierung der Diffusion hängen wir an alle Variablen der kontinuierlichen Version das Suffix “cont” und im Diskreten “disc” an. $ycont_c^k$ bezeichnet dann den kontinuierlichen Fluss auf einer Kante c, der von dem kontinuierlichen Algorithmus in der Iteration k ermittelt wurde.
Die diskrete Diffusion arbeitet in jeder Iteration wie folgt: Im ersten Schritt einer Iteration wird der kontinuierliche Fluss $ycont_c^k$ berechnet. Im nächsten Schritt versucht jeder Knoten diesen kontinuierlichen Fluss auf den inzidenten Kanten zu erfüllen. Hierfür sendet/empfängt er solange Prozesse, solange der kontinuierliche Fluss nicht überschritten wird. An dieser Stelle taucht ein Packungsproblem

[1] Dies kann nur bei der Diffusion mit Eigenwerten passieren, nicht aber bei der uniformen Diffusion.

auf, welches $\mathcal{NP}$-vollständig ist. Es müssen Prozesse ausgewählt werden, so dass der kontinuierliche Wert optimal angenähert wird. Dieses Problem verlagern wir, indem wir Fehlerlisten mitführen und der entstandene Fehler einer Iteration in der Folgeiteration berücksichtigt wird:

$$ydisc_c^k \leq ycont_c^k + e_c^{k-1} \qquad \text{mit} \qquad e_c^0 = 0 \tag{2}$$

In dieser Gleichung berücksichtigen wir den Fehler e_c^k der vorherigen Iteration, der sich wie folgt berechnet:

$${e_c}^k = ycont_c^k + e_c^{k-1} - ydisc_c^k \quad \forall c = \{\mathrm{n_i}, \mathrm{n_j}\} \in C \tag{3}$$

Um den Fehler der letzten Iteration zu minimieren, führen wir einen zusätzlichen Ausgleichsschritt durch:

$$e_c^m = e_c^{m-1} - y_c^{adj} \tag{4}$$

Die Güte der diskreten Diffusion im Vergleich zum kontinuierlichen Gegenstück, wurde in [9] untersucht.

3.2 Lokale Bipartitionierung

Der Algorithmus zur Bipartitionierung bestimmt zunächst das Verhältnis der Last einer HW- zu einer SW-Implementierung eines Prozesses: $w^H(p_i)/w^S(p_i)$. Anhand des Verhältnisses wählt der Algorithmus Prozesse aus und implementiert sie entweder in HW oder SW. Zusätzlich wird nur der Ressource ein Prozess zugewiesen, die weniger Last besitzt. Der folgende Quelltext zeigt die Arbeitsweise des Algorithmus:

```
foreach (new task pi on node nj){
  r[pi] = wH[pi]/wS[pi];
  // insert by increasing order in list
  r = insertList(r[pi]);
}

while (!taskList.empty()){
  if (totalSWload < totalHWload){
    // last task in the list r has maximal ratio
    task = popLastTask(taskList, r);
    totalSWload += task.SWload;
    SWtasks.addToList(task);
  }
  else {
    // first task in the list r has minimal ratio
    task = popFirstTask(r);
    totalHWload += task.HWload;
    HWtasks.addToList(task);
  }
}
```

Durch die konkurrierenden Zielgrößen kann es dazu kommen, dass Prozesse mit einem HW/SW-Lastverhältnis von größer als 1 als HW-Prozess implementiert werden, obwohl sie besser auf der SW-Ressource ausgeführt werden. Aus diesem Grund führen wir bei der Diffusion zusätzlich Prioritätslisten ein, so dass Prozesse, die suboptimal implementiert sind als erstes zu anderen Knoten migrieren. Die Laufzeitkomplexität des Algorithmus beträgt $\mathcal{O}(|P_i^{neu}|log_2(|P_i| + |P_i^{neu}|))$, was durch das Einsortieren neuer Prozesse in eine sortierte Liste zustande kommt. $|P_i|$ ist hierbei die Anzahl der Prozesse auf einem Knoten n_i und $|P_i^{neu}|$ die Anzahl der neuen Prozesse.

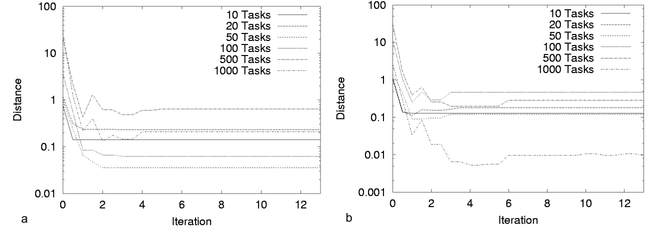

Abbildung 3. Distanz der Online-Partionierung für eine gegebene Anzahl an Prozessen

4 Experimentelle Evaluierung

Für die Evaluierung haben wir mit Hilfe eines Evolutionären Algorithmus, der globales Wissen über das Netzwerk besitzt, eine Menge nicht-dominierter HW/SW-Partitionen R ermittelt. Diese Ergebnisse wurden mit dem vorgestellten Ansatz zur lokalen verteilten HW/SW-Partitionierung verglichen, der eine Lösung s liefert. Die kürzeste Distanz $d(s)$ zwischen einem Punkt der Menge R und dem Punkt s gilt als Güte der Lösung:

$$d(s) = \min_{r \in R} \left\{ \left| \frac{s_1 - r_1}{r_1^{max}} \right| + \left| \frac{s_2 - r_2}{r_2^{max}} \right| \right\} \tag{5}$$

In den folgenden Experimenten haben wir ein 3x3-Mesh und einen Ring mit verschiedenen Lastverteilungen untersucht. Der Parameter s_1 bzw. r_1 ist hier die maximale HW- bzw. SW-Last auf den Knoten: $\max\{w_{n_i}^S, w_{n_i}^H\} \quad \forall n_i \in N$. Ist dieses Maximum minimal, hat man eine Balance zwischen den Knoten wie auch den HW/SW-Ressourcen auf einem Knoten erzielt, was den ersten beiden Zielgrößen aus Abschnitt 3 entspricht. Die Minimierung der Gesamtlast aus Abschnitt drei ist der zweite Parameter s_2 bzw. r_2.

Abbildung 3a) zeigt ein 3x3-Mesh, bei dem eine bestimmte Anzahl an Prozessen mit zufällig gewählten Lasten beliebig im Netzwerk verteilt sind. Die Anzahl der Prozesse variiert von 10 bis 1000 Prozessen im Netzwerk. In Abbildung 3b) sind die Ergebnisse der gleichen Prozessverteilung für eine Ringtopologie mit neun Knoten präsentiert.

Man kann deutlich sehen, dass der Algorithmus die initiale Normalverteilung von Prozessen wesentlich verbessert. Es wird allerdings auch deutlich, dass der Algorithmus in einem lokalen Minimum stecken bleiben kann, was daran liegt, dass er aus seiner lokalen Information auf kein globales Optimum schließen kann. In dem zweiten Szenario gehen wir von einem Netzwerk aus, welches sich in einem optimal balancierten Zustand befindet. Das heißt, dass alle Zielgrößen optimal erfüllt sind. Zusätzlich verdoppeln wir nun auf einem Knoten die SW-Last, was einem Szenario entspricht, in dem neue Prozesse in einem Netzwerk ausgeführt werden sollen. Ausgehend von diesem Zustand zeigt Abbildung 4a) und b) wie sich der vorgestellte Algorithmus a) für ein 3x3-Mesh und b) einen Ring verhält.

Für den beschriebenen Fall wurden verschiedene Lasten im Bereich von 100 bis 500 auf einem Knoten betrachtet. Ein

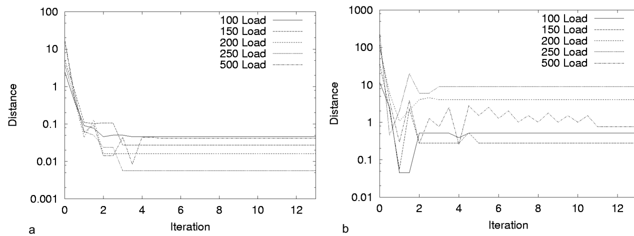

Abbildung 4. Distanz der Online-Partitionierung für eine gegebene Last auf Knoten.

einzelner Prozess kann eine nicht normierte Last im Bereich von 1 bis 20 auf einem Knoten verursachen. Man kann deutlich sehen, dass die Verdopplung der Last auf einem Knoten zu einem großen Fehler führt, der erneut von dem vorgestellten verteilten Algorithmus drastisch verringert wird. Alle Abbildungen zeigen übrigens die Distanz $d(s)$ nach jeder Bipartitionierung und nach jedem Diffusionsschritt.

5 Implementierung

Um die Anwendbarkeit der vorgestellten Methode zu untersuchen, wurde der Online-Partitionierungsalgorithmus für ein Netzwerk implementiert, das aus vier Altera-Excalibur-Boards besteht. Alle vier Boards können über serielle Punkt-zu-Punkt-Verbindungen miteinander kommunizieren. Sie beinhalten einen Cyclone-FPGA für HW-Prozesse und eine NIOS-CPU für SW-Prozesse.

Da unser Algorithmus für reale verteilte Anwendungen des Automotive- und Steuerungsbereichs gedacht ist, gibt es weitere Fragestellungen, die es zu Untersuchen gilt:

Dynamische HW Rekonfiguration: FPGAs wie der Virtex4 von Xilinx stellen zwar eine erste Möglichkeit zur partiellen Platzierung von HW-Modulen dar, dennoch ist Rekonfiguration, insbesondere partielle Rekonfiguration ein Thema derzeitiger Forschungsvorhaben [10].

Sichere Prozess-Migration: Für die sichere Migration von Prozessen in einem Netzwerk, ist es unumgänglich, dass Prozesszustände aus einem Knoten ausgelesen und zu einem anderen Knoten gesendet werden. Für SW-Prozesse gibt es viele Vorarbeiten zu diesem Thema und die Architekturen der gängigen Prozessoren erlauben eine Sicherung von Kontexten. Bei dedizierten HW-Prozessen muss man berücksichtigen, dass zur Entwurfszeit entsprechende Mechanismen und Datenstrukturen verwendet werden, die eine Sicherung des Zustandes erlauben.

Da bei den verwendeten Excalibur-Boards keine partielle Rekonfiguration möglich ist, hält jeder Knoten eine HW- und eine SW-Implementierung eines jeden Prozesses vor. In Abhängigkeit, ob ein Prozess in HW oder in SW implementiert ist, wird in dem aufgebauten Demonstrator ein “Enable-Flag” gesetzt, welches signalisiert, dass ein Prozess auf einem Knoten in HW bzw. SW ausgeführt wird. Für die Sicherung von Zuständen der HW-Prozesse beinhaltet jedes dedizierte HW-Modul eine Register-Kette, durch die der Zustand der HW extrahiert werden kann. Diese Werte können direkt in bestimmte Register der NIOS-CPU geladen werden.

6 Zusammenfassung und Ausblick

In diesem Beitrag haben wir die Online-Optimierung in eingebetteten Netzwerken betrachtet. Mit der vorgestellten Methode ist es möglich dynamische Lastanforderungen wie auch Ressourcefehler in einem Netzwerk zu kompensieren. Um dies zu erreichen, maximiert der vorgestellte Algorithmus die Lastreserven auf jeder Ressource auf der Basis von lokalen Informationen.

Neben bereits existierender Verfahren zur Fehlerbehandlung in zuverlässigen Systemen, stellt diese Methode einen weiteren Schritt zur Erhöhung der Fehlertoleranz dar. Durch die Online-Optimierung wird versucht, Lastreserven zu schaffen und die Fehlertoleranz der Systeme weiter zu erhöhen.

In zukünftigen Arbeiten soll der Algorithmus derartig angepasst werden, dass er selbständig terminiert. Ebenfalls spielt eine Analyse des Echtzeitverhaltens und die Adaptierung auf heterogene Netzwerke eine große Rolle.

Literatur

[1] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2002.

[2] G. C. Buttazzo and J. Stankovic. Adding Robustness in Dynamic Preemptive Scheduling. In *Responsive Computer Systems*, 1995.

[3] G. Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, Oct. 1989.

[4] R. Elsässer, A. Frommer, B. Monien, and R. Preis. Optimal and Alternating-Direction Loadbalancing Schemes. In *Proc. of Euro-Par 99, Parallel Processing*, pages 280–290, 1999.

[5] R. Elsässer, B. Monien, and R. Preis. Diffusion Schemes for Load Balancing on Heterogeneous Networks. *Theory of Computing Systems*, 35(3):305–320, May 2002.

[6] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 1997.

[7] M. López-Vallejo and J. C. López. On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):269–297, July 2003.

[8] R. Lysecky and F. Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Proceedings of the conference on Design, automation and test in Europe*, pages 480–485. IEEE Computer Society, 2004.

[9] T. Streichert, C. Haubelt, and J. Teich. Online and HW/SW-Partitioning in Networked Embedded Systems. In *Proc. of ASP-DAC'05 99*, page to appear, 2005.

[10] H. Walder and M.Platzner. A Runtime Environment for Reconfigurable Hardware Operating Systems. In *Proc. of FPL'04*, 2004.